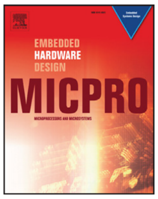
Microprocessors and Microsystems 39 (2015) 720–740

Contents lists available at ScienceDirect

Microprocessors and Microsystems

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



Real-time simulation of dynamic vehicle models using a high-performance reconfigurable platform

Madhu Monga, Daniel Roggow, Manoj Karkee[1], Song Sun, Lakshmi Kiran Tondehal, Brian Steward, Atul Kelkar, Joseph Zambreno*

Iowa State University, Ames, IA 50011, USA

ARTICLE INFO

Keywords:
FPGA
Real-time simulation
Hardware-in-the-loop
Non-linear functions
Hardware acceleration

ABSTRACT

With the increase in the complexity of models and lack of flexibility offered by the analog computers, coupled with the advancements in digital hardware, the simulation industry has subsequently moved to digital computers and increased usage of programming languages such as C, C++, and MATLAB. However, the reduced time-step required to simulate complex and fast systems imposes a tighter constraint on the time within which the computations have to be performed. The sequential execution of these computations fails to cope with the real-time constraints which further restrict the usefulness of Real-Time Simulation (RTS) in a Virtual Reality (VR) environment. In this paper, we present a methodology for the design and implementation of RTS algorithms, based on the use of Field-Programmable Gate Array (FPGA) technology. We apply our methodology to an 8th order steering valve subsystem of a vehicle with relatively low response time requirements and use the FPGA technology to improve the response time of this model. Our methodology utilizes traditional hardware/software co-design approaches to generate a heterogeneous architecture for an FPGA-based simulator by porting the computationally complex regions to hardware. The hardware design was optimized such that it efficiently utilizes the parallel nature of FPGAs and pipelines the independent operations. Further enhancement was made by building a hardware component library of custom accelerators for common non-linear functions. The library also stores the information about resource utilization, cycle count, and the relative error with different bit-width combinations for these components, which is further used to evaluate different partitioning approaches. In this paper, we illustrate the partitioning of a hardware-based simulator design across dual FPGAs, initiate RTS using a system input from a Hardware-in-the-Loop (HIL) framework, and use these simulation results from our FPGA-based platform to perform response analysis. The total simulation time, which includes the time required to receive the system input over a socket (without HIL), software initialization, hardware computation, and transfer of simulation results back over a socket, shows a speedup of 2 × as compared to a similar setup with no hardware acceleration. The correctness of the simulation output from the hardware has also been validated with the simulated results from the software-only design.

1. Introduction

Real-time simulation (RTS) is often a component of virtual prototyping used to study the dynamics of a physical system prior to actual hardware development. It has been utilized by engineers in various industries such as aviation [1], power systems [2], networking [3], automotive [4], traffic management [5], and medicine [6]. The physical systems encountered in these areas are mathematically modeled by deriving the ordinary differential equations (ODEs) that represent the underlying physics which dictate system behavior. To simulate these systems and estimate their state trajectories across a time duration, these ODEs are solved numerically using integration algorithms such as Runge-Kutta methods, Adams-Bashforth, or Adams-Moulton. The algorithms employ either fixed or variable integration time steps. The response generated by the simulation after each time step is considered useful for RTS only if the computation time for each time step remains below or equal to the actual time being simulated. However, the general-purpose CPU-based simulation of these systems continues to pose a major limitation on the smallest time-step with which RTS can be achieved. The reduced time-step required to simulate complex and fast systems imposes a tighter constraint on the time within which the computations have to be performed. The sequential

* Corresponding author. Tel.: +1 5152943312.
E-mail addresses: madhum@iastate.edu (M. Monga), dlroggow@iastate.edu (D. Roggow), manoj.karkee@wsu.edu (M. Karkee), sunsong@iastate.edu (S. Sun), kirantl@iastate.edu (L.K. Tondehal), bsteward@iastate.edu (B. Steward), akelkar@iastate.edu (A. Kelkar), zambreno@iastate.edu (J. Zambreno).
[1] Present address: Department of Biological Systems Engineering, Washington State University, USA.

http://dx.doi.org/10.1016/j.micpro.2015.08.014
0141-9331/© 2015 Elsevier B.V. All rights reserved.

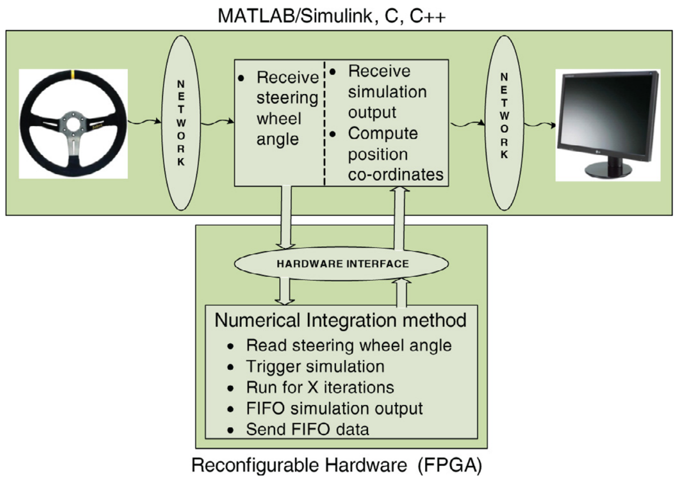

MATLAB/Simulink, C, C++
NETWORK
Receive steering wheel angle
Receive simulation output
Compute position co-ordinates
NETWORK
HARDWARE INTERFACE
Numerical Integration method
Read steering wheel angle
Trigger simulation
Run for X iterations
FIFO simulation output
Send FIFO data
Reconfigurable Hardware (FPGA)

Fig. 1. System architecture.

execution of these computations thus fails to cope with the real-time constraints which further restrict the usefulness of RTS in a Virtual Reality (VR) environment.

In this paper, we focus on acceleration of real-time (HIL) simulation of vehicle systems. In our target system, shown in Fig. 1 an operator provides an input via a physical steering wheel, and the steering input is then presented to the vehicle model, after which a graphical engine takes the output of the model and renders graphics showing the movement of the vehicle in a virtual world [7].

1.1. Background & motivation

The system of interest for this work is a steering valve subsystem of a vehicle model. This system is *dynamic*, since the rates of change within the system affect the response of the system to its environment [8]. In the steering valve system, for example, the rate at which the steering wheel is rotated can have significant ramifications on the direction of the vehicle. The general mathematical model for a continuous-time dynamic system is given in Eq. (1). The mathematical model for such a system must completely define the relationships between the system variables [8].

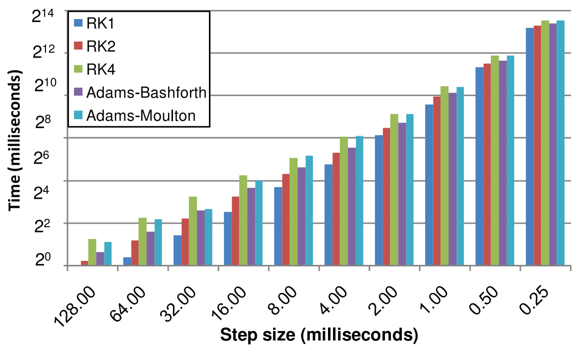
$$\frac{dy_i}{dt} = f(u_i, y_i) = A * y_i + B * u_i \qquad (1)$$

In the general form of the equation, y_i is an $N \times 1$ vector containing the state variables, which completely define the state of the system at an arbitrary instant of time [8]. The response (behavior) of the system is contained in these variables. The system inputs are contained in the $M \times 1$ vector u_i. Inputs to the system originate externally, and are largely independent of the behavior of the system [8]. A is an $N \times N$ state transition matrix that defines the coupling between various states of the system, and B is an $N \times M$ input matrix that relates the system inputs to the system states. A system is linear if all of the state and output equations are linear, and take the form of first-order differential equations. Note that this is different from the order of a model, which refers to the number of state variables present, e.g. the steering valve model described in the next subsection has 8 state variables, so is an 8th order model.

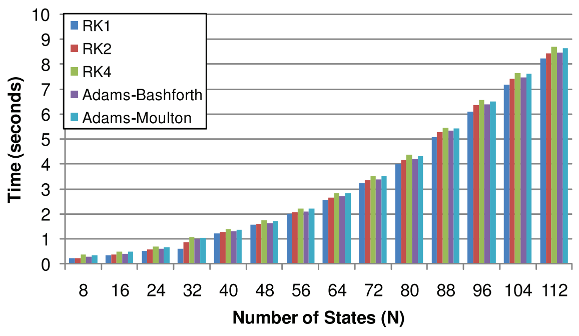
For an initial experiment, a simple and generic linear order model was implemented in MATLAB using different methods, where it was observed that the computation time was negatively affected by the increase in the number of computations involved in solving the ODEs, which varies with the choice of numerical integration method, the size of the time step t, and the number of physical states N being modeled. It is important to note that though an equivalent C implementation may be faster than a MATLAB implementation, we would still observe similar trends, although a more complex method and model and a smaller time-step would be needed.

Fig. 2 compares the CPU computation time with various time steps and number of states using different numerical integration methods. The computation time does not include the time required to receive the steering wheel angle, time to compute the position coordinates, and time to send these coordinates to the graphical engine. Fig. 2a shows the effect of reducing the time step on a 16th order linear system solved using RK1, RK2, RK4, Adams-Bashforth, and Adams–Moulton algorithms. As the time step is reduced, the time taken to simulate for five seconds increases for a fixed set of design parameters. Eventually, when the time step is reduced below 0.25 ms, the simulation fails to meet the real-time constraints for all the algorithms, and so further reductions in step size are not possible for this system. Fig. 2b shows the effect of increasing the number of states with a fixed time step of size 1 ms. When the number of states is increased to 88, all of the integration algorithms fail to meet the constraints, since the overall computation time surpasses the real time of 5 s. These results motivate our research into alternative platforms for simulating more complex vehicle dynamics in real-time.

In this work, we explore the reconfigurable capabilities of FPGAs for accelerating this class of algorithms and design a methodology to generate a reconfigurable architecture for different vehicle systems. FPGAs provide a platform to parallelize the independent computations and a custom pipelined architecture provides an opportunity to improve the throughput, though at the expense of an initial latency. More work per clock cycle results in a significant improvement in the computation time, and the reconfigurability allows the platform to be used for RTS of different vehicle systems without having to develop a custom ASIC for the same purpose. We aim to improve the end-to-end computation time for vehicle system simulation, which includes everything from the time the simulation model receives the



(a) Effect of time step using a 16th order linear system.



(b) Effect of number of states using a time step of 1 ms.

Fig. 2. Effect of step size and number of states on the CPU computation time for a linear system integrator.

user input to when the hardware sends the simulation results back to the display monitor.

The vehicle system targeted for hardware implementation in this work consists of two subsystems. The first component is a steering valve subsystem containing the dynamics of a hydraulic system that uses the steering input to compute the steering angle of the front wheels of the vehicle. The second component is the vehicle subsystem relating the steering angle to the trajectory of the vehicle as it is propelled forward at a constant velocity. To simulate the system in real-time, an integration time step of 10 μs is required for the valve subsystem simulation and a time step of 2 ms is required for the vehicle subsystem. A CPU-based (MATLAB) simulator was first used to simulate the whole vehicle system. However, it was observed that the computation time taken to run the simulation was 13 μs per integration time step, which fails to meet the 10 μs requirement. Fig. 4a further describes the main motivation behind our use of FPGA technology to implement the RTS of the vehicle system. It compares the computation time of the vehicle system for a simulation period of five seconds on an FPGA running at 55 MHz, and the single-threaded MATLAB simulation using the RK4 integration method running on a 2.83 GHz Intel Core 2 Quad CPU. The computation time increases with an increase in the number of states of the system for both the implementations. However, for the CPU-based simulator, the computation time exceeds the real time when the number of states in the system model is greater than 88. We can intuitively say that a more complex system with additional subsystems and forces will further add to the required computation time per iteration and result in the violation of constraints, even with a fewer number of states.

On the other hand, the computation time for the FPGA-based simulator remains well below the real-time constraint, since the fine-grained parallelism present in the model can be extracted using FPGAs, which would not be as beneficial using a multicore CPU or general purpose graphics processing unit (GPGPU). Additionally, the real-time constraints for this problem make GPGPUs and heterogeneous CPU/GPU platforms unsuitable as a targetable solution platform, due to the inherent uncertainty of task scheduling present in such throughput-oriented architectures [9,10]. Hence, we target an FPGA platform in this work.

1.1.1. 8th order vehicle system – steering valve and vehicle model

The steering valve dynamics model is this work uses a gerotor motor and a rotary-valve assembly to direct the fluid to different branches of a double-ended cylinder. The four valve openings on the cylinder, two on the left and two on the right, are used to direct flow to and from the cylinder. The hydraulic dynamics of the rotary-valve assembly is based on establishing a relationship between the pressure at four different volumes—two in the two sides of the gerotor motor and two in the two ends of the cylinder—and the net flow rate (through different valves) to hydraulic volume, given by Eq. (2) [8].

$$\dot{p} = \frac{\beta}{V} \times (\Delta Q_v) \tag{2}$$

where $\dot{p}$ is the pressure, β is the bulk modulus, ΔQ_v is the net flow rate to volume, and V is the total volume.

The four valves control the flow rate through four openings of the cylinder. The valve opening area is a function of relative displacement ($rdel$) between the angular position of the steering wheel (A_s) and the gerotor motor (A_m) given by

$$rdel = A_s - A_m \tag{3}$$

A_s is obtained from the continuously changing steering wheel input from the HIL and A_m is computed using the formula

$$A_m = \frac{-q_1}{I_g} \tag{4}$$

where q_1 is the derived state of the system and I_g is the gerotor inertia. Since the valve opening area is a function of two dynamically changing values A_s and A_m, the flow rate through these valves also changes continuously and is computed using the relation given below:

$$sqrt = \sqrt{\left(\frac{2 \times abs(p_i - p_f)}{\rho}\right)}$$
$$Q = A(\Theta) \times C_d \times sqrt \times sign(p_i - p_f) \tag{5}$$

where $A(\Theta)$ is the valve opening area, p_i and p_f are the inlet and outlet pressure at the valves, C_d and ρ are the constants that define the coefficient of discharge and the fluid density, respectively.

The vehicle dynamics of the system is governed by the displacement of a cylinder piston from its neutral position which, in turn, is controlled by the flow rate through valves. The angular displacement of the piston thus forms the system input for the vehicle model. The details of the dynamics of the vehicle model are explained in [7].

Fig. 12 shows the architecture of the vehicle system. The steering valve model consists of three components: the valve opening area unit, the orifice flow rate unit, and the state-space solver. The vehicle model consists of two components: a trigonometric function and a state-space solver. The non-linear dynamics of the steering valve model reads A_s from the HIL and previous state of the valve to compute the system input for the state-space solver. The solver implements a numerical integration method, with a time step h_{valve}, to compute the new state of the valve. The state variables model different attributes of the valve, and one of the states is used to compute the system input for the vehicle model using a trigonometric function. The solver for the vehicle model also implements the numerical integration method with a time step $h_{vehicle}$. It reads the previous state of the vehicle and the system input to compute the new state of the vehicle which is sent to the display monitor.

The valve opening area unit updates the valve opening area of all the valves at every time step. For each valve, the maximum and minimum relative displacement between A_s and A_m is fixed. We divide this range into equally spaced values and compute the corresponding area and thereby obtain a look-up table that holds the valve opening area for predefined relative displacement values. Using a linear-interpolation method, we can compute the valve opening area for any value between the given maximum and minimum relative displacements.

The orifice flow rate unit updates the flow rate through each valve. It reads the opening area of the valve $A(\Theta)$, available at the output of the valve opening area unit, along with the present state y_i of the system and computes the flow rate through each valve using Eq. (5). The inlet and outlet pressure values at the four valves are determined from four of the eight states in y_i. The flow rate through the four valves (and two dummy valves which do not affect the state of the system), constant pump outlet pressure (P_p), and A_s constitute the system input vector for the valve model.

One of the eight states of the steering valve model tracks the piston displacement position after every time step, h_{valve}. The trigonometric function is used to convert the linear displacement to the angular displacement of the piston that eventually forms the system input for the vehicle model.

We assume the valve model receives input A_s every S seconds. To determine the final state of the vehicle after S seconds, the steering valve model is simulated for S/h_{valve} iterations, followed by the simulation of the vehicle model for $S/h_{vehicle}$ iterations. The input of the vehicle simulation is either the output of the last iteration, or the average of all the iterations over S seconds of the steering valve model. The output of the vehicle model on the last iteration is the state of the vehicle after S seconds.

The state-space solver for both of the models performs the actual simulation process by numerically integrating the models at their respective time steps. These models are in the general form of the state-space representation of a linear system, as given in Eq. (1).

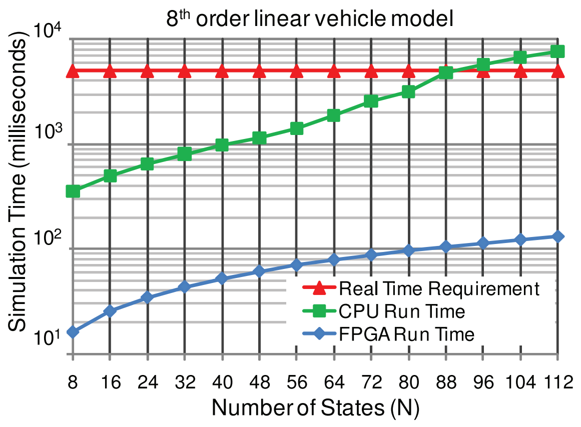
$$A = \begin{bmatrix} \frac{\beta_1(C_{L1})}{V_1} & \frac{-\beta_1(C_{Lm}+C_{L1})}{V_1} & \frac{\beta_1(C_{Lm})}{V_1 * I} & \frac{-\beta_1(V_d)}{V_1 * I} & \frac{-\beta_1(V_d)c^2}{V_1 * I^2} & 0 & 0 & 0 \\ \frac{\beta_2(C_{L2})}{V_2} & \frac{-\beta_2(C_{Lm})}{V_2} & \frac{-\beta_2(C_{Lm}+C_{L2})}{V_2 * I} & \frac{-\beta_2(V_d)}{V_2 * I} & \frac{-\beta_2(V_d)c^2}{V_2 * I^2} & 0 & 0 & 0 \\ \frac{-\beta_3(C_{L3})}{V_3} & \frac{-\beta_3(C_{Lc})}{V_3} & \frac{-\beta_3 A_C}{V_3} & 0 & 0 & 0 & 0 & 0 \\ \frac{\beta_4(C_{L4})}{V_4} & \frac{-\beta_4(C_{Lc})}{V_4} & \frac{\beta_4 A_C}{V_4} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & \frac{A_c}{m} & \frac{-A_c}{m} & \frac{-k_1}{m} & \frac{-c_1}{m} & 0 & 0 \\ V_d & -V_d & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & \frac{-c_2}{I_g} \end{bmatrix} \quad y_i = \begin{bmatrix} p_1 \\ p_2 \\ p_3 \\ p_4 \\ x \\ v \\ q_1 \\ q_2 \end{bmatrix}$$

(a) Coefficient matrix A and state variable vector y_i

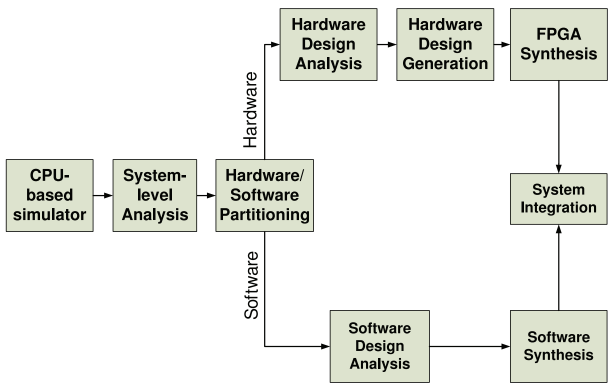
$$B = \begin{bmatrix} 0 & \frac{\beta_1}{V_1} & \frac{-\beta_1}{V_1} & 0 & 0 & 0 & 0 & \frac{-\beta_1(V_d)c^2}{V_1 * I^2} \\ \frac{\beta_2}{V_2} & 0 & 0 & \frac{-\beta_2}{V_2} & 0 & 0 & 0 & \frac{-\beta_2(V_d)c^2}{V_2 * I^2} \\ 0 & 0 & \frac{\beta_3}{V_3} & 0 & 0 & \frac{-\beta_3}{V_3} & 0 & 0 \\ 0 & 0 & 0 & \frac{\beta_4}{V_4} & \frac{-\beta_4}{V_4} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -k_2 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -c_2 \end{bmatrix} \quad \vec{u} = \begin{bmatrix} Q_{ol1} \\ Q_{or1} \\ Q_{ol2} \\ Q_{or2} \\ Q_{ol3} \\ Q_{or3} \\ P_p \\ A_s \end{bmatrix}$$

(b) Coefficient matrix B and input variable vector u_i

Fig. 3. Coefficients and variables for steering valve and vehicle model.



(a) FPGA vs CPU vs real-time simulation with different number of states using a time step of 1 ms



(b) General hardware/software co-design approach for vehicle system simulation

Fig. 4. Motivation and approach.

Although the original form of the steering valve model is non-linear, the required coefficient matrices and vectors for a linear state-space representation of the model were generated using the relation in Eq. (2) [11].

The coefficient matrix (A) and state variable vector (y_i) for the steering valve model are shown in Fig. 3a. In A, the first four rows model the hydraulic dynamics, where β_1 to β_4 are the fluid bulk moduli of the volumes V_1 to V_4. C_{L1}, C_{L2}, C_{L3}, and C_{L4} are the leakage-flow coefficients for each respective volume. C_{Lm} is the gerotor motor leakage-flow coefficient, C_{Lc} is the cylinder leakage-flow coefficient, and C_p is the flow-pressure coefficient for the pipe. The corresponding state variables are given by p_1 to p_4.

The next two rows in A model the cylinder piston dynamics, and the variables are: the cylinder area A_c, the cylinder viscous damping c_1, the cylinder spring constant k_1, and the gerotor motor moment of inertia I. The state variables are represented by the velocity (v) and position (x) of the cylinder piston.

The last two rows in A represent the gerotor motor and rotary valve assembly dynamics: the gerotor frictional damping c_2, the valve centering spring constant k_2, the gerotor displacement V_d, and the equivalent mass m of the of steering system. The state variables representing the derived states are q_1 and q_2.

The coefficient matrix (B) and system input vector (u_i) for the vehicle model are shown in Fig. 3b. In the input vector, Q_{ol1}, Q_{ol2}, and Q_{ol3} are flow rates through the left end of the cylinder, and Q_{or1}, Q_{or2}, and Q_{or3} are flow rates through the right end of the cylinder. The rows corresponding to flow rates Q_{ol3} and Q_{or3} in the input matrix B are zeros, meaning these flow rates do not affect the final state of the steering valve system, ultimately reducing the number of pressure values in the state variable vector for the four valves from 6 to 4.

1.2. Paper contributions

We propose a hardware/software co-design approach to accelerate the RTS using a heterogeneous parallel architecture. Fig. 4b provides an overview of our approach. Using this approach, we claim the following contributions to the state-of-the-art in simulation of vehicle system dynamics which otherwise fail to meet the real-time constraints using a software (CPU-based) simulator:

- A co-design approach for RTS by partitioning the tasks between a hardware and a software platform
- A methodology based on an heuristic approach to generate an FPGA-based simulator. The approach uses a hardware component library which contains fast hardware implementations of non-linear functions and timing information of these components
- Application of our methodology to generate the FPGA-based simulator for the vehicle system and various design strategies explored based on our methodology
- Proof-of-concept of RTS using a simulator with both hardware and software components

2. Related work

Ever since their genesis in the mid-1980s, FPGAs have been used in various fields for prototyping, acceleration and reconfigurable computations. [12] outlines the benefits of FPGA implementations in various fields and the advantages of such reprogrammable systems. Initially, FPGAs were either used to emulate ASIC-targeted applications to test the design before the production of custom hardware [13], or to accelerate computationally intensive applications which would otherwise have poor performance when implemented in software. For example, temporal pattern and speech recognition using a hidden Markov model first compares the digital voice signals with the English language phonemes to generate a search string. The search string is then compared with the dictionary words for the closest match. As the size of the dictionary grows, matching becomes computationally intensive. The parallel nature of FPGAs enhances the search process by allowing simultaneous execution of the independent portions of the design [14]. Other computationally intensive applications that have been successfully used in parallel architectures on the computational fabric of FPGAs are graph problems, such as finding a Hamiltonian cycle [15], mathematical methods, such as the finite-difference time-domain method [16] and Jacobi iteration [17], and, finally, communications decoding algorithms [18].

FPGAs have also played a significant role in the performance improvement of molecular dynamics (MD) algorithms. MD simulates the motion and interaction between atoms or molecules based on different forces acting between these particles. Computing these forces using a single processor solution is very slow. In 2004, [19] was the first work published which tested the feasibility of MD using FPGAs. [20–22] further demonstrate the usefulness and performance improvement of FPGA-based MD simulations. FPGAs can also be used in bioinformatics. The earliest such use of hardware acceleration for biological sequence comparison was in 1998, with a dedicated hardware accelerator, called SAMBA (Systolic Accelerator for Molecular Biological Applications) [23]. [24] achieved a speedup of 200 $\times$ over the conventional desktop implementation for protein sequence alignment, and [25] showed a speedup of 383$\times$ for the multiple DNA sequence alignment problem by implementing the computationally intensive part of the comparison algorithm on an FPGA. Because of the significant progress in the field of FPGA-based acceleration, financial modeling methods have also been ported to FPGA implementations. [26–28] demonstrate a speedup of up to 80 $\times$ for the Monte Carlo simulation algorithm. [29] uses FPGAs for portfolio management. FPGA-based Gaussian distribution models, which are used to model correlation between different entities, such as between portfolios containing hundreds of assets, achieved a speedup of 33 $\times$ over a CPU-based implementation [30].

The use of FPGAs for accelerating real-time simulations of systems has recently increased. [31,32] implemented a real-time simulation for permanent magnet synchronous motors using the RT-LAB real-time simulation platform, as well as auto generated hardware blocks from Xilinx Simulink Generator (XSG). FPGAs have successfully been used to study the dynamic behavior of large power systems through manual hardware design of a high frequency power system simulator. [2] explored a hardware-software codesign approach for dual time step real-time simulation of power systems. High frequency transient phenomena in power systems can also be simulated on FPGAs. [33] proposed an FPGA-based real-time electromagnetic-transient program simulator which is capable of simulating systems with a time-step of 12 μs, which is well below 50 μs, the minimum required time step for transient simulations. FPGAs thus offer a promising platform for the simulation of fast transients in power systems.

Minimal work has been done on accelerating vehicle system dynamics using FPGAs. Recently, [34] demonstrated real-time simulation of railway-vehicle dynamics using an FPGA-based accelerator. A fast MATLAB/Simulink implementation of a railway-vehicle simulator completed each 1 ms time step in 21.5 ms, whereas the FPGA implementation only consumed 0.625 ms per time step. However, the time does not include the communication delay due to the distributed simulation architecture. The rest of this paper details a hardware/software codesign approach for accelerating real-time simulations of vehicle dynamics.

3. Design methodology

In the system-level analysis of Fig. 4b, we determine the partitions based on two factors: the computation time of different components of the simulation model, and the frequency of communication between different components. To efficiently utilize both the hardware and software resources, we obtain an initial partition such that the computation-intensive part of the simulation model, and modules which can benefit the most by the parallel architecture, are implemented on the hardware, and the rest are implemented in software. If there is continuous exchange of data between the two partitions, the increased communication delay between hardware and software will negatively affect the overall computation time. The reason for leaving part of the system in software is twofold: if all of the system were placed into hardware, a significant amount of hardware resources would be consumed, and the design may not fit onto the target platform. Secondly, the methodology presented in this paper focuses on only accelerating in hardware the necessary portions of the system in order to meet the designer's real time constraint for simulation. Software not as critical to the performance of the system, such as the input controller, can remain in software without adversely affecting the real-time constraint, which makes maintenance on those portions easier. Additionally, it is important to note that the methodology presented in this paper is not attempting to achieve the overall optimal performance possible for a hardware/software system, but to improve the performance only as much as is necessary in order to meet the real-time constraint as provided by the system designer. With these issues in mind, we first discuss the hardware partitioning followed by software partitioning.

3.1. Factors affecting hardware partitioning

Hardware partitioning is governed by three factors: accuracy/precision in the simulation results, space occupied on the hardware, and the time required to complete the computations of a single time step. The accuracy/precision and hardware resource utilization (RU) are affected by the manner in which the data is represented on the hardware. For this work, we use fixed-point representation [35] which consists of a fixed number of integer and fractional bits before and after the fixed-point. The accuracy is determined by the number of integer (I) bits available, whereas the precision is governed by the number of fractional (F) bits selected for fixed-point representation. As the number of bits is increased to improve accuracy/precision in the simulation results, the space required for the hardware implementation also increases.

The relation between time and space is based on the parallelism that can be exploited in the FPGA-based simulator. If all the independent computations of the CPU-based design are implemented concurrently, then the resulting FPGA-based simulator would complete a single iteration in the shortest possible time. However, the parallelism comes at the expense of hardware resources. Were the computation done serially, each component would begin execution only after the previous component finishes, and the total computation time would be the longest. In this case, the hardware RU will be equivalent to that of the single largest component. A pipelined implementation of the design will result in increased throughput, and the hardware RU will be somewhere between that of the earlier two implementations.

Intuitively, the partitioning between hardware and software for this type of system follows a pattern that can be extracted into a design suitable for a particular system. For example, the underlying model in the real-time vehicle simulation system presented in this

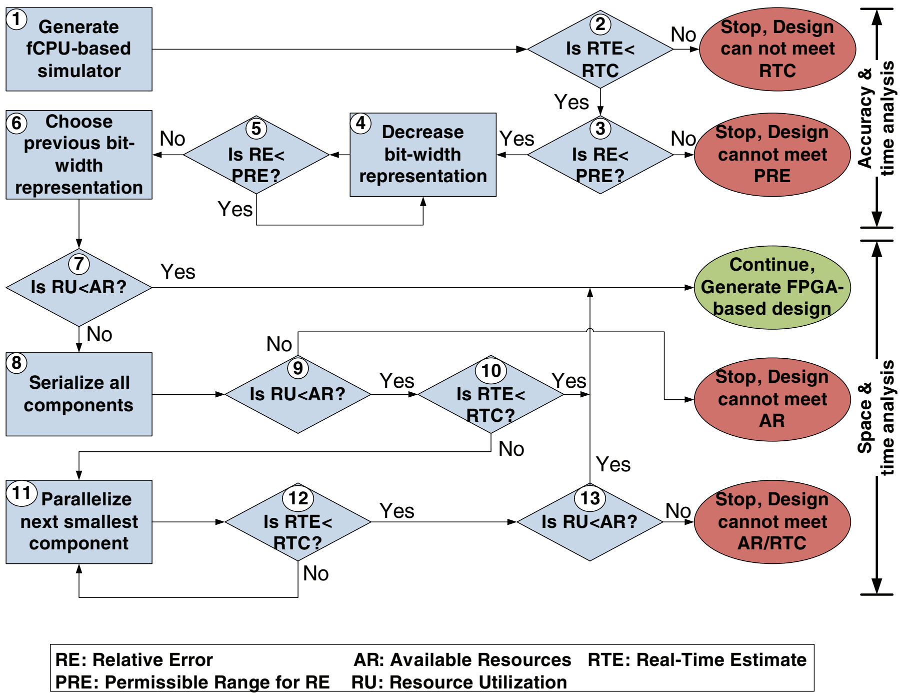

1 Generate fCPU-based simulator
2 Is RTE< RTC
No
Stop, Design can not meet RTC
Yes
3 Is RE< PRE?
No
Stop, Design cannot meet PRE
Yes
4 Decrease bit-width representation
5 Is RE< PRE?
Yes
No
6 Choose previous bit-width representation
Accuracy & time analysis
7 Is RU<AR?
Yes
Continue, Generate FPGA-based design
No
8 Serialize all components
9 Is RU<AR?
No
Yes
10 Is RTE< RTC?
Yes
No
Stop, Design cannot meet AR
11 Parallelize next smallest component
12 Is RTE< RTC?
Yes
No
13 Is RU<AR?
Yes
No
Stop, Design cannot meet AR/RTC
Space & time analysis
RE: Relative Error
AR: Available Resources
RTE: Real-Time Estimate
PRE: Permissible Range for RE
RU: Resource Utilization

Fig. 5. Heuristic approach for hardware partitioning.

paper is, for the most part, linear, and such models have well-known parallelizable algorithms to compute solutions. Other equations necessary for the full model definition can be broken down into constituent mathematical functions and operators, which can be implemented using a library of mathematical operations. These individual pieces can then be joined together to form a hardware implementation of the computationally intensive portions of the system.

The methodology to generate the FPGA-based simulator, based on the factors discussed above, is divided into three phases: hardware design analysis, hardware design generation and verification, and software design analysis. Fig. 5 shows the heuristic approach for the hardware design analysis phase where we analyze the requirements (i.e. the required bit combination, the time taken to complete a single iteration, and the hardware RU). The hardware design generation phase uses this information to generate the actual hardware design.

3.2. Hardware design analysis

The input to the hardware design analysis phase is the CPU-based simulator which uses the RK4 integrator, Permissible Relative Error (PRE) in the simulation output, the Real-Time Constraint (RTC), and the available hardware resources (AR). The input is based on the assumption that the platform is decided in advance, with an aggressively parallelized design and the maximum number of bits for fixed-point representation. Ideally, the PRE value should be set by the engineers who design the simulation model; they should be able to determine the acceptable relative error in the simulation results.

3.2.1. Accuracy/precision and time analysis

Step 1: To implement the methodology described in Fig. 5, we need a model which can give us an estimate of the required bit combination, the time taken to complete a single iteration, and the hardware RU. These estimates can be obtained by having a model that can emulate the FPGA computation process which we call a fixed-point CPU-based (fCPU-based) simulator. We first design a software component library containing equivalent software (MATLAB) representations of all the components in the hardware component library. Each function is implemented using the same techniques that are used in the hardware design. The functions in the CPU-based simulator are then replaced with their modified implementation from the software component library. For example, instead of using the MATLAB built-in `ode45` function for integration, we implement the algorithm for RK4 in MATLAB and use the same algorithm implementation for the matching hardware component. In addition, the arithmetic operations in the modified implementation are also done using fixed-point notation and are parameterized for different bit combinations. Since the algorithms used are the same as those used for hardware implementation, the architecture is close to that of the FPGA-based simulator. The computation process also emulates the FPGA computation, thus making the fCPU-based simulator an appropriate model to estimate the required bit combination that affects the accuracy/precision and the hardware RU.

Step 2: Before we actually generate the design, we estimate whether the hardware is capable of even meeting the RTC with a completely parallelized design. A parallelized design assumes that all the independent computations are implemented in parallel, optimizing for time. Thus, the time obtained from such a design is the estimate of the minimum possible time which the hardware will take to compute the output of a single iteration.

The fCPU-based simulator gives us the hardware components required for the FPGA-based simulator. For each of the independent components in the hardware component library, we use the cycle information to compute the number of cycles for the entire design to generate the output, accounting for any parallelism used in the design. Assuming different hardware clock frequencies, we can determine the Real-Time Estimate (RTE) of this design for these clock frequencies. The RTE is compared against the minimum possible time that an FPGA-based simulator will take by exploiting all the parallelism in the model. If the RTE is more than the input RTC, the hardware will not be able to meet the RTC. If the time taken remains within the RTC, we check for the Relative Error (RE) constraint in the next step.

Step 3: In Section 3.1, we discussed that the number of bits affect the accuracy/precision with which the data values and computation results are represented on the hardware. To obtain an estimate of the

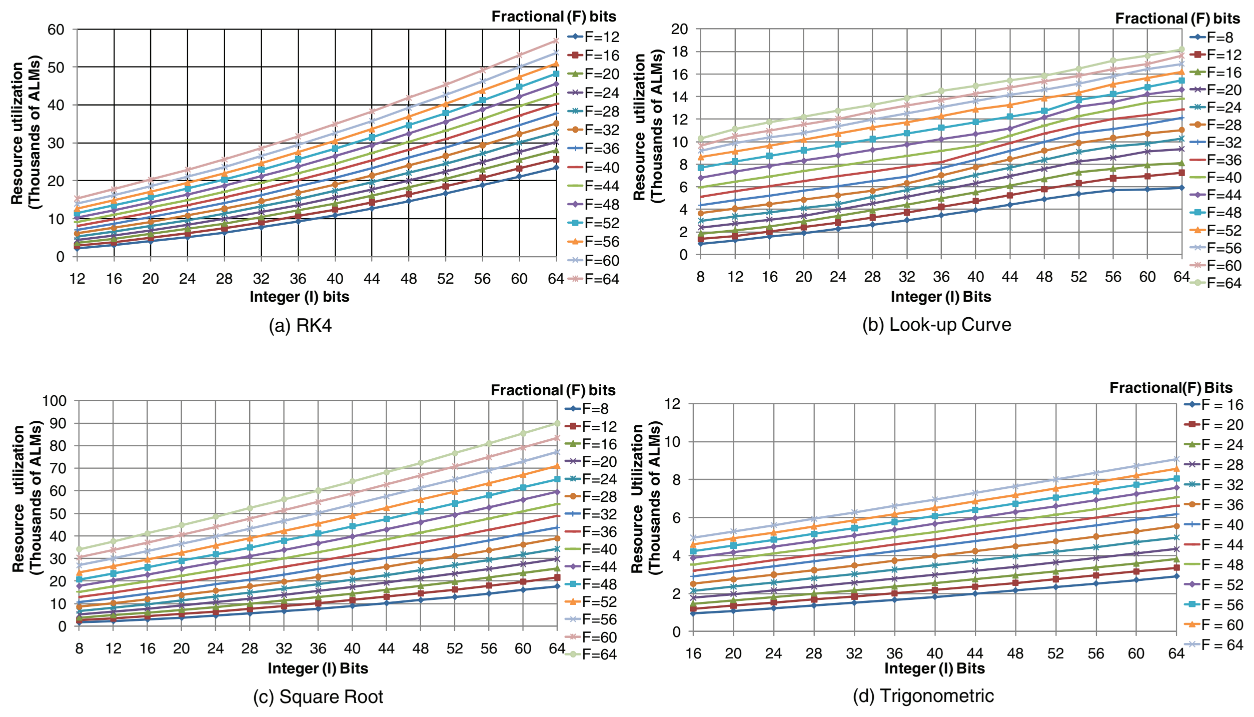


Fig. 6. Hardware resource utilization for RK4, look-up curve, square root and trigonometric components.

required bit combination without generating the FPGA-based simulator, we emulate the hardware computation process in the fCPU-based simulator. This is achieved by converting all the data values at each computation step into the fixed-point format of length $M = I + F$ bits. The converted values are equal to, or close to, the true values if the number of bits is sufficient. The local truncation error due to each conversion and the global propagation error due to previous conversions result in an error in the final simulation output after every iteration. Since the fCPU-based simulator is generated using algorithms used for the FPGA-based simulator, the conversion accurately models the hardware computation process, and the error generated from this process can be considered a close estimate of the RE that will be generated from the FPGA-based simulator. If the maximum number of representation bits produce a RE outside of the PRE constraint, no suitable hardware partition exists.

Steps 4, 5 and 6: If the constraints are met, we further optimize the design by reducing the bit-width combination such that the RE remains within the PRE. We first reduce the number of I bits while keeping $F = 64$. After obtaining the number of sufficient I bits, we reduce the number of F bits until it fails the constraint. However, the process of estimating the bit-combination using the fCPU-based simulator is highly dependent upon the number of iterations the simulation is run. As we increase the number of iterations, the number of sufficient bits that satisfy the PRE criteria may increase, so the selected bit-width combination may not be the final estimate that would represent the values close to the required values on the hardware.

3.2.2. Space and time analysis

Initially, the timing analysis is performed assuming an aggressively parallelized model, which, if implemented on hardware, would utilize the maximum available resources. If the optimized design meets the RTC, it can then be optimized for space to determine if the model meets the AR constraints.

On the hardware, as the RU increases, the area covered by the design increases and so does the path traversed by the clock. This in turn lowers the overall frequency at which the design can run. During space analysis, we thus optimize the design for space by serializing or pipelining the components. However, optimizing for space increases the time taken to run a single iteration.

Before we proceed to the next step, we present our approach to estimate the hardware RU of the design. The components present in the hardware component library are independent entities that can be plugged into any design, as long as the input and output ports are correctly mapped. We ran the hardware synthesis for all the components in the library and obtained their hardware RU for different bit combinations. The results are detailed in Fig. 6. The synthesis was run on Altera's Stratix III board, so the RU is in terms of Altera's Adaptive Logic Modules (ALMs). An equivalent number of 6-input LUTs on Xilinx Virtex-5 FPGAs can be obtained using the relation given in [36].

Step 7: For space analysis, we first check if the design meets the AR constraint using the bit-width combination selected in Step 6. We use the fCPU-based simulator to determine the components that make up the FPGA-based simulator and use the graphs for resource utilization (for some example components) to determine their RU for the selected combination. We compare the RU for the whole design with the AR input for the selected platform. If the constraint is met, we use the automated scripts to generate the VHDL-design for the selected components with the selected bit-width combination. If it does not, we perform the space optimization and start with a completely serialized design in Step 8.

Steps 8, 9 and 10: In Step 1, we started with an aggressively parallelized design (optimized for time), to check for the RTC and obtain an estimate of the speed-up that can be achieved. To optimize for space, we serialize/pipeline the components such that a new computation starts either after completion of the previous computation or a cycle later. This process reduces the RU, since the number of computations done in parallel is reduced. However, to obtain a lower limit on the hardware RU of the design, we start with a completely serialized design using the selected bit-width combination and check if this optimized-for-space design is able to meet the AR constraint (Step 9). If the AR constraint cannot be satisfied, then it is impossible to proceed any further, since the serialized design has the lowest RU. If the constraint is satisfied, we then check if the new design meets the RTC (Step 10). As mentioned earlier, the serialization affects the RTE, and, if the completely serialized design meets the RTC, we use automated scripts to generate the VHDL design for the selected components with the selected bit-width combination. If the RTC is not satisfied, subsequent steps attempt to find a blend of parallelized and serialized components that meet the constraints.

Steps 11, 12 and 13: In Step 8, we made the assumption of a completely serialized design and obtained the minimum amount of resources consumed by such a design. Since the RTC is not met, while still optimizing for space, we parallelize the component which consumes the minimum resources and thus results in the minimum increase in the overall RU. We check for the RTC in Step 12, and iterate through Step 11 and Step 12 until we meet the RTC. As discussed earlier, as components are parallelized, the RTE decreases but the RU increases. Once the RTC is satisfied, the RU constraint is checked. If the RU constraint still fails, it is impossible to generate an FPGA-based simulator fulfilling the necessary requirements.

To meet the real-time constraints, the simulation results should be available at the host for further processing before the RTC is exceeded. This is necessary because the time taken to complete an iteration includes both the computation time and the communication delay. Consequently, the RTE is the sum of the computation time and a communication delay that varies with the amount of data transferred. Therefore, it is necessary to have an interface that provides sufficient bandwidth and minimizes the latency of data transfer between the host and the hardware. Since the hardware/software partitions have already been decided, the amount of data exchanged between the two partitions is known. This information is used to determine the required bandwidth of the interface and compare it with the bandwidth of the selected platform.

3.3. Hardware design generation and verification

An important aspect of the design methodology is automatic generation of the design models based on different design decisions. The design decisions in this case include the selection of an appropriate bit combination that meets the accuracy, time, and space criteria. Such automation is advantageous for designers because it allows them to focus on making the best design decisions without devoting the significant amount of time required for re-creating designs whenever a change is necessary.

3.3.1. Design generation

The VHDL design for each component is highly parameterized and pipelined. The parameters for each component, and those specific to vehicle system simulation, are saved as constants in the parameters file in fixed-point format based on the bit combination. To use these constants, the components need to include the parameters file while implementing the design. However, the selection of an appropriate bit combination is an iterative process during which the parameters and the VHDL design have to be regenerated. For a complex system with numerous components, this step would require the designer to re-create the design for all of the components every time the bit combination changes. Therefore, to automate the process of design generation, we designed MATLAB scripts to take the bit combination and order of the system (if required) as input, and then generate the parameters file and VHDL design for the components, including any serialization or pipelining.

3.3.2. Design verification

In this step, we compare simulation output from Modelsim with that from the CPU-based simulator. Since there are different components connected together, it is essential to validate that data from these components is represented correctly. Assuming the design meets the functionality criteria, an insufficient number of bits may result in a mismatch of the final simulation output if the output from any component is incorrect. After analyzing whether the mismatch is due to an insufficient number of I or F bits, we increase the bits accordingly and go back to Step 7 of the hardware design analysis phase. In addition to data validation, simulation is an important phase to check the speedup that might be expected from the present implementation. Once the Modelsim simulation shows a perfect match with the results from the CPU-based simulator, we generate the programming file and integrate it with the software design to run the RTS. At this point, it is no longer necessary to run both the CPU-based simulator and the Modelsim hardware simulator, unless a need to debug the hardware simulator arises later in the design process.

Table 1
Cycle count by individual hardware components.

Component	Cycle count
RK4 vehicle	113
RK4 valve	169
Look-up curve	5
Square root	12
Trigonometric	7

3.4. Software design analysis

The software design analysis phase is based on the platform selected because the width of the interface governs the alignment and data format in which the system input should be sent to the hardware, and the simulation output should be received back from the hardware. With focus on vehicle system simulation, the software design should be able to perform the following tasks for the complete HIL RTS:

- Receive the system input from the HIL
- Send the system input to the hardware
- Receive the simulation output from the hardware
- Convert the hexadecimal format of the output to the decimal format
- Perform software computation, if any
- Send the simulation output to the VR display.

If the system input is assumed to change every T ms, the software should be able to perform the above tasks, which includes communication delays and hardware computation time within this amount of real time. The fast computations on the hardware can allow the simulation to complete before the equivalent real-time interval elapses. Thus, to emulate the real-time scenario, we start the timer in the software just before it receives the system input. The hardware runs the simulation for T ms and sends the output back to the software and stalls until it receives the new system input. On the software side, the timer stops after sending the simulation output for further processing. At this point, if the difference between the stop and start timer is less than T ms, a sleep command is invoked to stall the software for the remaining amount of time (i.e. $T - (stop - start)$).

As the complexity of the simulated physical system increases, the amount of workload for both the hardware and software partitions also increases. To satisfy the RTC, it is essential to develop an efficient software design that minimizes the duration between the start and stop timers.

4. Hardware component library

To apply the methodology discussed in Section 3, we developed a hardware component library of functions required for the hardware implementation. Included in our hardware library are components for look-up curve, square root, and trigonometric functions, as well as a RK4 integrator. Table 1 shows the computation time for each component in number of clock cycles.

Table 2
Comparison of Xilinx and Altera square root IP cores with our custom square root core.

	Xilinx Virtex 7		Altera Stratix IV	Altera Stratix III
	CORDIC [37]	Our work[a]	Floating point [38]	Our work[b]
Data Type	Fixed point	Fixed point	Single precision	Fixed point
Latency[c] (cycles)	48	2	28	2
I/O Width	48 bits	128 bits	23 bit mantissa	128 bits
Xilinx resource/Altera resource				
LUT6 FF pairs/ALUTS	2714	3092	502	2985
LUTs/DLRs	2500	3080	932	632
FFs/ALMs	2511	913	528	1780
Block rams/Memory blocks	0	0	0	0
DSP48E1/DSP18	0	196	–[d]	272
Max clock (MHz)	283	55	472	55

[a] Synthesized with the Xilinx tool chain v14.7.
[b] Synthesized with the Altera tool chain v12.1.
[c] Steady state.
[d] Altera does not provide this value.

The interfaces for each component are very similar to each other, and to the overall system interface. The whole system behaves similarly to a FIFO queue, and so each component has data input and data output signals, as well as several control signals to notify components upstream and downstream of data ready or data needed events.

4.1. Rationale

One method of computing trigonometric and square root functions is to use coordinate rotation digital computer (CORDIC) algorithms. Xilinx takes this approach [37] for their trigonometric and square root IP core. This core uses fixed point representation and provides two configurations for implementation: parallel and word serial. A parallel implementation consumes more resources than word serial, but can produce a result every clock cycle after an initial latency. The word serial implementation can only work on one result at a time, so has lower throughput.

Another approach is to compute the result using floating point numbers, and this seems to be the approach Altera uses [38]. Although Altera is sparse on details, floating point implementations in general tend to have increased complexity and resource usage than fixed point implementations.

A last approach is to simply use a look-up table and couple it with a linear interpolation method or use the value as a starting place for a computation. There are several reasons for selecting a look-up table approach, including reduced complexity, improved latency and throughput, and portability between FPGA vendors. However, the most compelling reason to use a custom look-up table for our system is because of our special input data width requirements. Our design requires fixed point inputs of size 128 bits (I and F can each have a maximum of 64 bits), but Xilinx only supports input widths of up to 48 bits. To overcome these limitations, we developed our own components for non-linear functions which support 128-bit input widths. Table 2 compares our custom square root component with Xilinx and Altera solutions as a reference. Note that Altera does not provide the number of DSPs used by their floating point design.

4.2. Look-up curve component

4.2.1. Principle

The look-up curve component estimates the value of a function at a point *input* given the value of the function at two precise data points X_1 and X_2. It reads the function values at known data points from a look-up table and uses the linear interpolation method to obtain this estimate, which is given by the equation below.

$$Estimate = \left(\frac{Y_2 - Y_1}{X_2 - X_1}\right) \times (input - X_1) + Y_1 \qquad (6)$$

where the *input* lies between X_1 and X_2, and Y_1 and Y_2 are the corresponding function values.

For the hardware implementation, we split the look-up table into two tables such that one contains the slope estimate as given by

$$Slope = \left(\frac{Y_2 - Y_1}{X_2 - X_1}\right) \qquad (7)$$

and the other table contains the corresponding Y values. As we will see later, we do not need to store X values, except for the minimum, X_{min}, and maximum, X_{max}, since they can easily be computed. X_{min} and X_{max} are saved in the parameters file. For any given input, the two X values between which the *input* lies is computed using the formula given below:

$$index = \left\lfloor \frac{input - X_0}{\Delta X} \right\rfloor \qquad (8)$$

where ΔX is the difference between the equally spaced X values, $input - X_0$ determines how far the *input* is from the first X value, X_0. Given three known values (i.e. *input*, X_0, and ΔX), we can compute the *index*. We use the same formula to compute X_1 by replacing *input* with X_1. The *index* is used to compute the read *address* for both tables.

4.2.2. Implementation

In a five stage pipelined architecture, shown in Fig. 7, a valid computation starts when a *start* signal is received. In the first stage it checks if the input is within the maximum and minimum range of the X values. It also computes the *index* using Eq. (8). In the second stage, the index is pipelined for one more stage and the integer bits of the index are converted to an unsigned format to compute the *address* for reading the two tables. In the third stage, while the memory is being accessed, we compute X_1 using the formula given below, and also compute the difference $input - X_1$.

$$X_1 = X_0 + \Delta X \times index \qquad (9)$$

In the fourth stage, we have all the necessary values to compute Eq. (6). The value of $input - X_1$ is available from the previous stage, and *Slope* and Y_1 are read from the memory. The final output, *Estimate*, is pipelined for one more stage and is available at the end of the fifth stage. The pipelined implementation can accept a new *input* value every cycle. After an initial latency of 5 cycles, our component generates a new output every cycle.

A two-cycle delay in the computation of the *Estimate* is avoided by splitting the look-up tables into two. Furthermore, the implementation is highly parameterized, such that all the parameters which do not change during the simulation are saved as constants in the parameters file. The division by a constant value in Eq. (8) is implemented by saving the inverse of ΔX in this file. When the bit

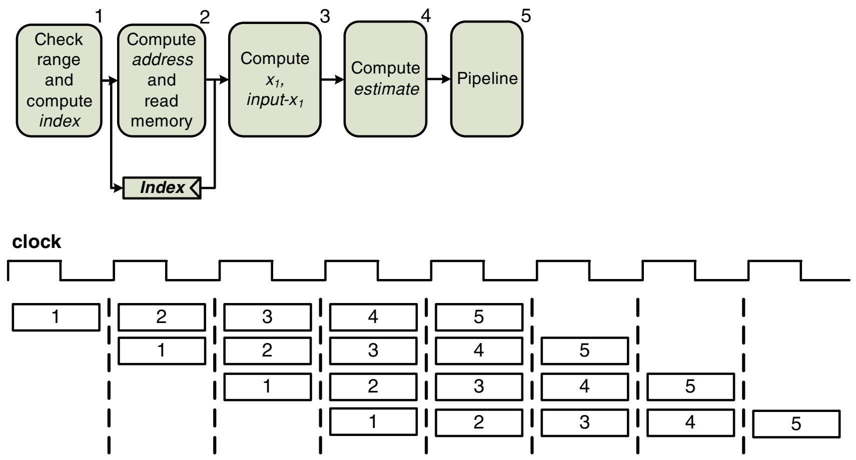

Check range and compute *index*
Compute *address* and read memory
Compute x_1, *input-x_1*
Compute *estimate*
Pipeline
Index
clock

Fig. 7. Pipeline of look-up curve component.

Table 3
Estimated resource utilization of look-up curve component on an Altera Stratix III FPGA.

Resource	Amount
ALUTs	913
DLRs	641
ALMs	759
DSP18s	54

combination changes, the auto-generation scripts for the VHDL design regenerate these constants and look-up tables in the fixed-point format for the new combination. The estimated resource utilization for this component is shown in Table 3.

4.3. Square-root component

The algorithms to compute the square root in hardware fall into two categories: subtractive and multiplicative [39–41]. Subtractive, or direct methods, are based on the conventional procedure of computing the square-root by hand, where each bit of the result is computed in one clock cycle. This method is efficient for a small number of input bits, but the initial latency is very high for a larger number of input bits. The multiplicative methods (Newton-Raphson and Goldschmidt algorithms) on the other hand, iteratively refine the initial approximation to compute the square-root. Though the algorithms exhibit a quadratic convergence, they are expensive in terms of resource utilization. Since our focus is on acceleration of the RTS, where speed is of prime importance, we chose the latter category for our implementation. The Newton-Raphson method contains dependencies between its successive operations, causing an uneven pipeline structure, thus, we use the Goldschmidt algorithm [42].

4.3.1. Principle

The square root function is implemented using the Goldschmidt algorithm [42] which is efficient in computing the square root of values close to 1. We base our idea on the fact that any number can be represented in the form $2^n \times a$, where n is an integer and a is a number close to 1. 2^n is the largest power of 2 that appears in the number, the square root of which is obtained from a look-up table that holds precomputed square root values. The square root of a is determined using the Goldschmidt algorithm. The square root of the original number is the product of the two square root values. The Goldschmidt algorithm is a three-step process, described in Eqs. (10)–(12) where x_0 and y_0 are set to the initial guess value a. When the process is executed for several iterations, as x_i converges to 1, y_i converges to $\sqrt{a}$.

$$r_i = (3 - x_i)/2 \tag{10}$$

$$x_{i+1} = x_i \times r_i \times r_i \tag{11}$$

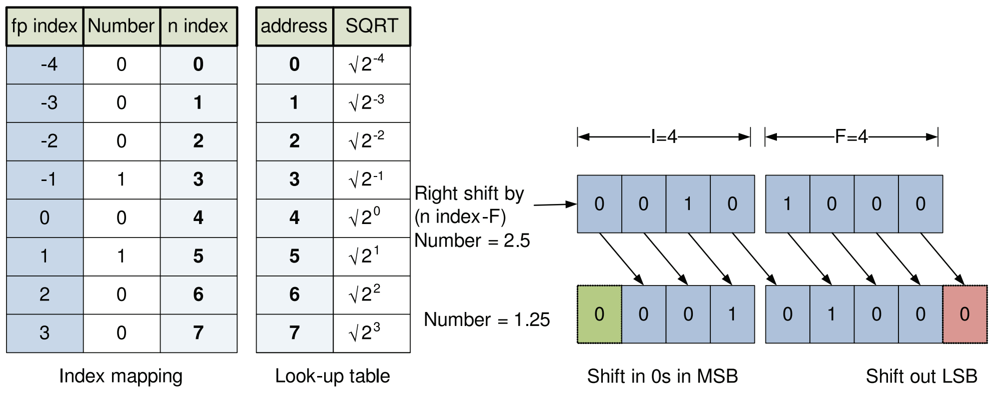
$$y_{i+1} = y_i \times r_i \tag{12}$$

4.3.2. Implementation

The look-up table implementation is based on the bit combination selected for the FPGA implementation. Given the number of I and F bits, the maximum and the minimum number that can be represented as a power of 2 are 2^{-F} and 2^{I-1}, respectively. The look-up table stores the square root of the following numbers: 2^{-F}, 2^{-F+1}, $2^{-F+2}, \ldots, 2^0, \ldots, 2^{I-3}, 2^{I-2}, 2^{I-1}$. To show that the look-up table returns the correct square root value, we introduce 2 index values: the normal (n) index and the fixed-point (fp) index. The normal index is the index interpreted by the hardware for any binary number and also the address at which to read the look-up table. The fixed-point index is the index interpreted for fixed-point arithmetic and also the index of the numbers whose square root values are stored in the look-up table.

Fig. 8 explains the methodology to obtain the number 2^n, its square root, and a for computing the square root when *number* $= 2.5$ and with a bit combination of $I = 4$ and $F = 4$. In Fig. 8a, the table on the left-hand side gives binary representation of the number, along with index n and index fp values. The table on the right-hand side is the look-up table generated for the selected bit combination. To determine the largest power of 2 which is close (and can be represented with the given bit combination) to the *number*, we check index n and index fp corresponding to the first occurrence of 1 in the most significant bit (MSB). These values are 5 and 1, respectively. The look-up table at address 5 stores the square root of 2^1, which is the largest power of 2 that appears in the *number*. Fig. 8b depicts generating a number close to 1, by shifting the *number* such that the first occurrence of 1 in the MSB now lies at the 0th position of the I bits. The number of bits to shift is computed by subtracting F bits from the index n value. If the input is less than 1, index n will be less than F bits and the negative difference will shift the input left. A positive difference will shift the input right. In this case, *number* is shifted right by $5 - 4 = 1$ bit.

During the first stage of the pipelined architecture, we compute the address to access the memory where the look-up table is stored. The address is pipelined in order to be used in the later stages to read the memory. The initial guess value, a, for the Goldschmidt algorithm is also computed in this stage. r_i is computed in the second



fp index	Number	n index
-4	0	**0**
-3	0	**1**
-2	0	**2**
-1	1	**3**
0	0	**4**
1	1	**5**
2	0	**6**
3	0	**7**

address	SQRT
0	$\sqrt{2^{-4}}$
1	$\sqrt{2^{-3}}$
2	$\sqrt{2^{-2}}$
3	$\sqrt{2^{-1}}$
4	$\sqrt{2^{0}}$
5	$\sqrt{2^{1}}$
6	$\sqrt{2^{2}}$
7	$\sqrt{2^{3}}$

(a) Determine 2^n and its square root (b) Determine a

Fig. 8. Methodology to obtain 2^n, its square root and a.

stage. The computation of x_{i+1} and y_{i+1} occurs in the third stage. The Goldschmidt algorithm is executed for five iterations, resulting in a total runtime of 10 cycles. Since the output from the Goldschmidt algorithm is not available until end of the 11th stage, a valid read enable signal is sent to the memory in the 10th stage, along with the pipelined address value computed in the first stage. By the end of the 11th stage, we receive the square root of 2^n from the memory, the square root of a from the algorithm, and compute the product of the two in the final stage. Therefore, the output is available at the end of the 12th stage. With this pipelined architecture, after an initial latency of 12 cycles, a valid square root value can be obtained every other cycle. The estimated resource usage is shown in Table 2.

4.4. Trigonometric function component

4.4.1. Principle

The trigonometric function is implemented using the linear-interpolation method described in Eq. (6). However, the implementation is slightly different from the one used for the look-up curve component. With the linear-interpolation method, a better approximation of the function can be achieved when the interval between the two precise data points is as small as possible. Though the approximated value gets close to the actual value, an increased number of data points consumes more FPGA resources. The estimated resource usage for this component is similar to that of the look-up curve component shown in Table 3.

We use the same architecture to compute both the trigonometric and inverse trigonometric functions. However, the difference is in the way the input and output data values are interpreted. To compute a trigonometric function, we explore the symmetry among the function values and generate a look-up table that contains the sine values for 1250 equally spaced points between the interval 0 to $\pi/2$. The cosine value is generated from the same table using the identity $\cos(x) = \sin(\pi/2 - x)$, and the input is modified accordingly. Also, the function values in other quadrants can be computed using trivial trigonometric math, but they may have a different sign. To determine the sign, we first reduce the input to the component to be in the range 0 to 2π, by computing the floor of the value obtained by division of the number with 2π. The obtained integer result is multiplied with 2π and then subtracted from the original number. The result is the reduced number in the required range. The component determines the quadrant of the input and the sign, depending on whether it is computing sine or cosine, and reduces the input to the range 0 to $\pi/2$. To compute the inverse trigonometric function, we generate a look-up table that contains the inverse sine values for 1250 equally spaced points between the interval 0 to 1. The cosine value is generated from the same table using the identity $\arccos(x) = \pi/2 - \arcsin(x)$.

For a look-up table approach with 1250 points, the principle described in Section 4.2, would need two such tables. If the number of bits selected is small, it would be possible to implement a two table approach; however, with a large number of bits, the resource usage would be tremendous. Thus, we use a single table approach, which adds a 2 cycle delay to the computation time of the look-up curve described in Section 4.2.

We make two assumptions for the input that is sent to this component for computing the trigonometric function. Firstly, it is always a positive value. Secondly, it is in the range of 0 to 2π. A number greater than 2π is converted to a number within this range. For a negative input value, we compute the two's complement of the number and send the modified value to the component. For the sine function, we restore the sign by taking the two's complement of the output, since $\sin(-x) = -\sin(x)$. For a cosine function, $\cos(-x) = \cos(x)$, so a second two's complement operation is not required. For inverse trigonometric functions, the input is saturated in the range between 0 and 1.

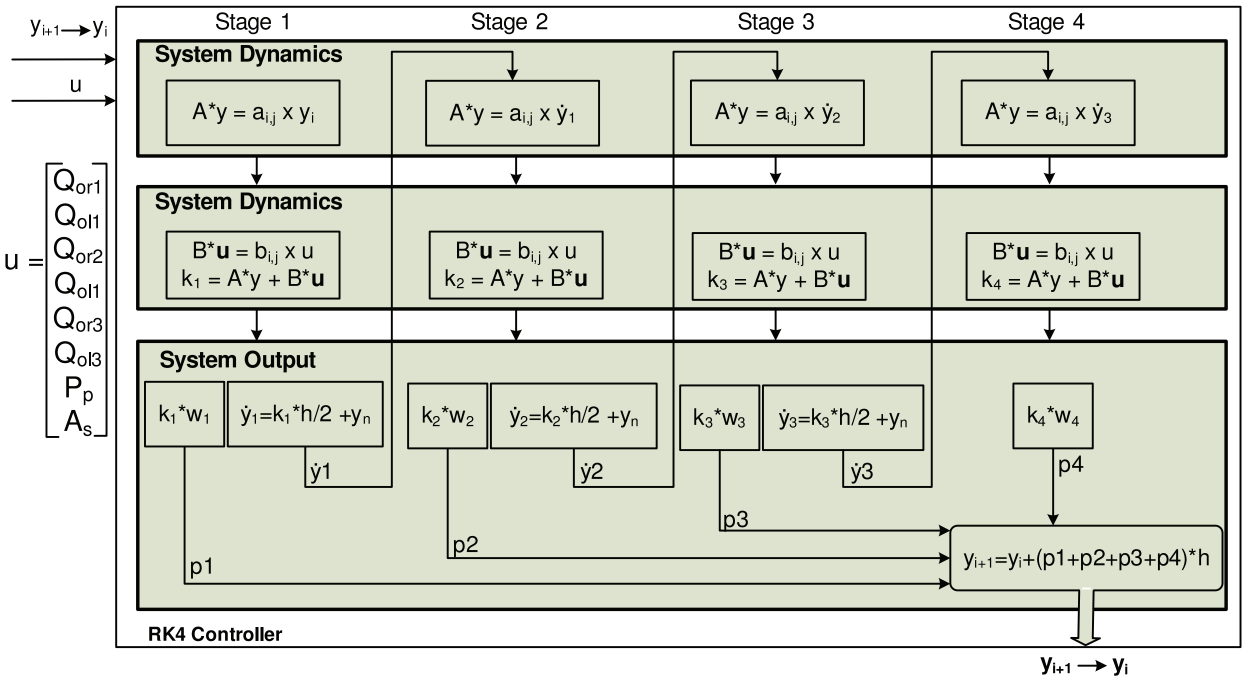
The component is fully pipelined to generate a new trigonometric function value every cycle after an initial delay of seven cycles. For computing inverse trigonometric functions, a reduction of the input to the range 0 to $\pi/2$ is not required. So, in the first stage, we simply register the input and compute the index. However, in the first stage for computing trigonometric functions, a reduction to the required range is done before computing the index. In the second stage, the index from the first stage is used to compute the address of two consecutive data points. The third stage does the same arithmetic as the look-up curve from Section 4.2.2. In the fourth stage, instead of computing Eq. (6), we compute the *difference* of the two function values read from the memory, and also pipeline the lower index data point until the sixth stage. In the fifth stage, the division of *difference* by the constant ΔX is implemented by multiplying with its inverse to obtain the slope estimate. In the sixth stage, the function value is obtained by computing the product of the result from the fifth stage, the slope estimate, with the result from the third stage, $input - X_1$, and adding the result to the lower index data point read from the memory. The output is then pipelined for one stage and made available at the end of the seventh stage.

4.5. State-space solver - RK4 integrator

The actual simulation process is performed by solving the state-space representation of the system using the RK4 integration method [43], and the steps involved are depicted in Fig. 9. The function f is the state-space representation of the system described in Eq. (1), and we use this representation to formulate the dynamics of the steering valve and vehicle model. The steps involved in the computation of

RK1	RK2	RK4
$y_{i+1} = y_i + k_1 * h$ *where* $k_1 = f(u_i, y_i)$, y_i = State of the system at t_i, y_{i+1} = RK1 approximation of $y(t_{i+1})$, h = Time Step	$y_{i+1} = y_i + (k_1 + k_2) * h / 2$ *where* $k_1 = f(u_i, y_i)$, $k_2 = f(u_i + h, y_i + k_1)$, y_i = State of the system at t_i, y_{i+1} = RK2 approximation of $y(t_{i+1})$, h = Time Step	$y_{i+1} = y_i + (k_1/2 + k_2 + k_3 + k_4/2)*h / 3$ *where* $k_1 = f(u_i, y_i)$, $k_2 = f(u_i + h/2, y_i + k_1/2)$, $k_3 = f(u_i + h/2, y_i + k_2/2)$, $k_4 = f(u_i + h, y_i + k_3)$, y_i = State of the system at t_i, y_{i+1} = RK4 approximation of $y(t_{i+1})$, h = Time Step

Fig. 9. Numerical integration using Runge–Kutta methods.



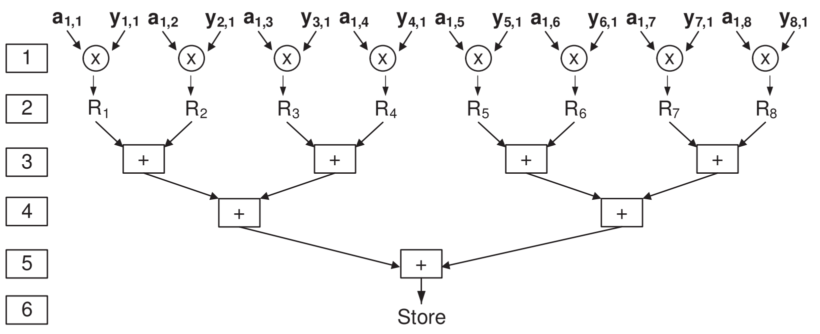
$y_{i+1} \rightarrow y_i$
u
u =
Q_{or1}
Q_{ol1}
Q_{or2}
Q_{ol1}
Q_{or3}
Q_{ol3}
P_p
A_s
Stage 1
Stage 2
Stage 3
Stage 4
System Dynamics
$A*y = a_{i,j} \times y_i$
$A*y = a_{i,j} \times \dot{y}_1$
$A*y = a_{i,j} \times \dot{y}_2$
$A*y = a_{i,j} \times \dot{y}_3$
System Dynamics
$B*\mathbf{u} = b_{i,j} \times u$, $k_1 = A*y + B*\mathbf{u}$
$B*\mathbf{u} = b_{i,j} \times u$, $k_2 = A*y + B*\mathbf{u}$
$B*\mathbf{u} = b_{i,j} \times u$, $k_3 = A*y + B*\mathbf{u}$
$B*\mathbf{u} = b_{i,j} \times u$, $k_4 = A*y + B*\mathbf{u}$
System Output
k_1*w_1
$\dot{y}_1=k_1*h/2 + y_n$
k_2*w_2
$\dot{y}_2=k_2*h/2 + y_n$
k_3*w_3
$\dot{y}_3=k_3*h/2 + y_n$
k_4*w_4
$\dot{y}1$
$\dot{y}2$
$\dot{y}3$
p4
p3
p2
p1
$y_{i+1}=y_i+(p1+p2+p3+p4)*h$
RK4 Controller
$y_{i+1} \rightarrow y_i$

Fig. 10. Data-flow diagram for RK4 iteration.

y_{i+1} are sequential, since the computation of the new slope (k_2, k_3, k_4) is dependent on the previous slope (k_1, k_2, k_3). Since we cannot parallelize the computations of y_{i+1}, we explore the parallelism within each computation. The RK4 integrator for the FPGA-based simulator consists of three components: the RK4 controller, the system dynamics, and the system output. The last two components implement the actual dynamics of the integrator, and the first component implements a state machine to control the flow of information between the other two components. As shown in the data-flow diagram walking through a single iteration (Fig. 10), each iteration is divided into four stages. During the four stages, the system dynamics computes the slope estimate k_l (l is the stage number) and feeds the result to the system output component. The system output computes the intermediate state, $\dot{y}_l$ (except in the last stage), which is fed to the system dynamics. It also adds and pipelines the intermediate results of the weight and slope multiplication. In the fourth stage, the system output adds these pipelined results and computes the new state of the system.

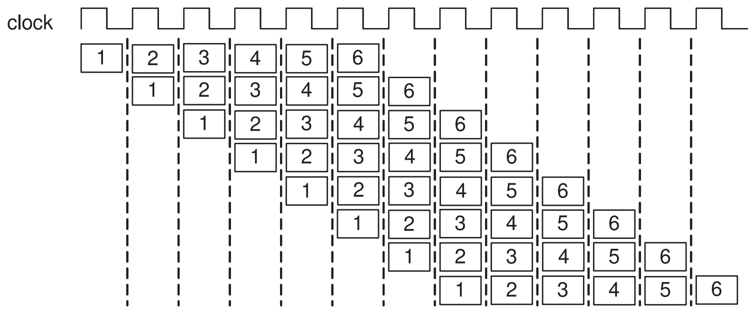
We use the parallel architecture of the FPGAs to pipeline computations within the components in each stage. The system dynamics implement matrix-vector multiplication, which is equivalent to z independent vector–vector multiplications, where z is the number of rows in a matrix. The architecture of the FPGA gives us the ability to do these computations in parallel. However, the amount of work that an FPGA can do in one clock cycle is limited by its clock frequency. For example, in an Nth order model, a N-by-N matrix is multiplied with a N-by-1 vector, where each element is $I + F$ bits long. The total number of operations required in one clock cycle are N^2 multiplications and $N \times N - 1$ additions. As the matrix size, or the number of bits, increases, it gets more difficult for the FPGA to perform all these computations in one clock cycle. Therefore, we pipeline the matrix-vector multiplication such that a new vector-vector multiplication is issued every cycle. The number of multiplications issued is equal to the number of rows z in the matrix. Fig. 11a shows different stages in the vector-vector multiplication of $a_{(1,\,1:\,8)} \times y_{(1:\,8,\,1)}$ for an 8th order system. Fig. 11b shows the number of cycles taken by the pipeline architecture to compute $A \times y$. The matrix-vector multiplication for $B \times u$ follows a similar architecture and can be implemented in parallel with $A \times y$. However, to improve the timing characteristics of the design, we serialize them.

In the six stage pipelined architecture for the system dynamics component, the first stage is used to compute a vector-vector multiplication. In the next stage, the required number of bits from the product are saved in the zeroth vector of a 2D array. In the third stage, the eight product terms are added using four adders and the result is saved in the first vector location of the 2D array. In the fourth stage, we add these four data values and save the result in the second vector location. The two data values are summed in the fifth stage. An



a1,1 y1,1 a1,2 y2,1 a1,3 y3,1 a1,4 y4,1 a1,5 y5,1 a1,6 y6,1 a1,7 y7,1 a1,8 y8,1
1
2
3
4
5
6
R1 R2 R3 R4 R5 R6 R7 R8
Store

(a) Pipeline stages to compute $a_{(1,1:8)} \times y_{(1:8,1)}$



(b) Pipeline architecture to compute $A \times y$

Fig. 11. Pipeline architecture of System Dynamics component for an 8th order system.

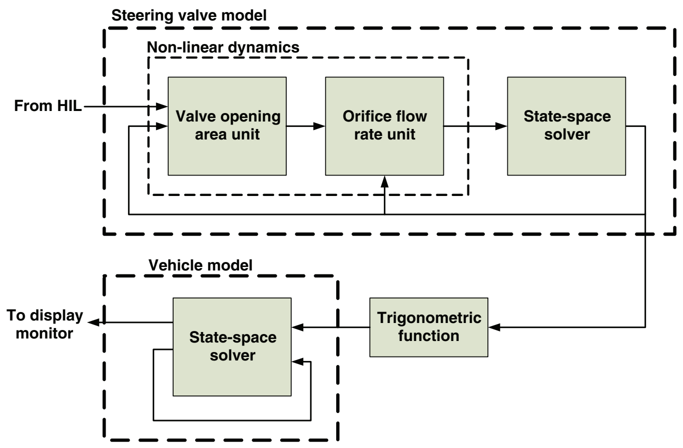


Fig. 12. Architecture of the vehicle system.

additional stage is used to pipeline the result. For an 8th order system, each of these stages is implemented for the eight vector-vector multiplications, so 13 cycles are consumed to compute $A \times y$, as shown in Fig. 11. The multiplication of $B \times u$ also goes through the same stages, but with an extra stage to sum the result of the two matrix-vector multiplications, which brings the cycle count to 14 for the second matrix-vector multiplication. The system output takes 12 cycles to compute intermediate $\dot{y}_l$ states, and the RK4 controller, which controls the state transitions, takes another 13 cycles to change states between the two components. Since there are 4 stages, the total number of cycles is $4 \times 14 + 4 \times 13 + 4 \times 12 + 13 = 169$ clock cycles.

An optimization of this implementation is when the system input u is scalar instead of a vector quantity. In this case, matrix B is a vector of size $N \times 1$, and $B \times u$ is reduced to a vector-vector multiplication. This optimization allows the parallel implementation of the two multiplications, $A \times y$ and $B \times u$, and also reduces the number of state transitions. The RK4 controller now takes 9 cycles instead of 14, because the two multiplications can be done in parallel. The total number of cycles is reduced to $4 \times 14 + 4 \times 12 + 9 = 113$ clock cycles. Table 4 shows the estimated resource utilization for this component.

Table 4
Estimated resource utilization of RK4 componcolruent on an Altera Stratix III FPGA.

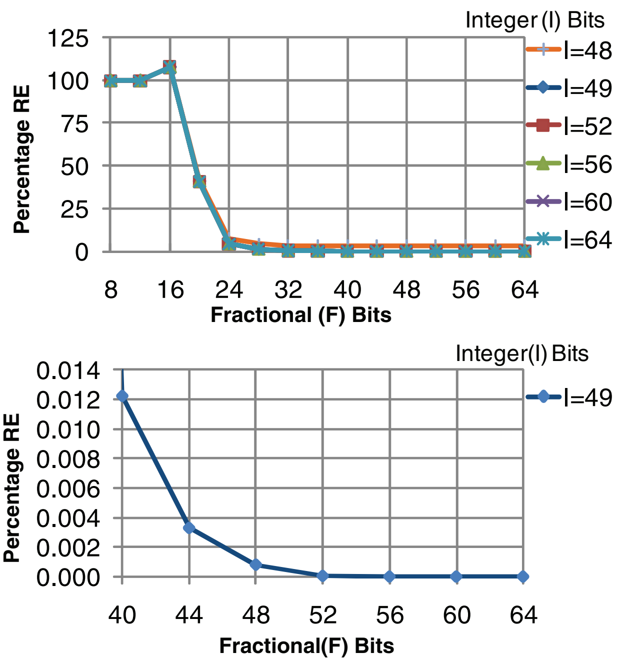
Resource	Amount
ALUTs	3508
DLRs	6410
DSP18s	120

5. Application of the methodology

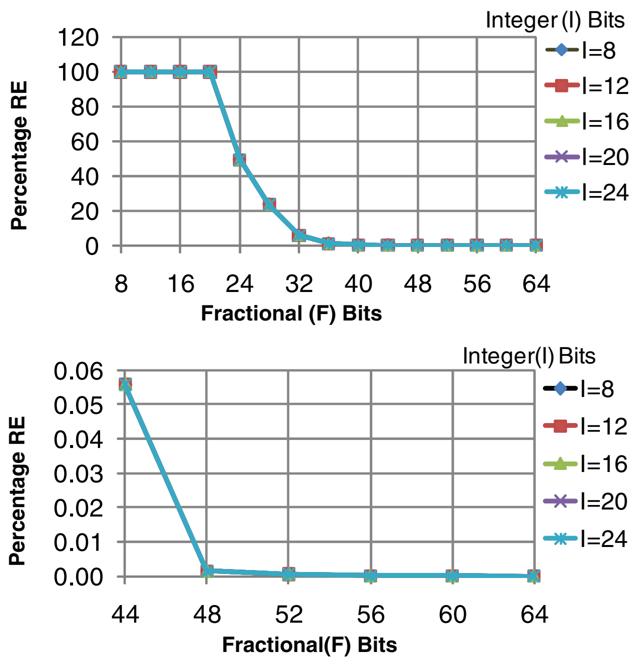
We use the library of components described in Section 4 to generate an FPGA-based simulator for an 8th order steering valve subsystem of a vehicle. The steering valve dynamics, described in detail in [11], simulates the dynamics between the rotation of the steering wheel and the rotational motion of the front wheel about the king pin. The dynamics of this system are quite stiff due to small time constants associated with internal volumes in the steering valve. Thus, with RK4, the system is numerically unstable for integration steps larger than time step h_{valve} of 10 μs. The vehicle subsystem dynamics [7] that describe the vehicle motion based on steering angle inputs can be simulated with a time step $h_{vehicle}$, on the order of a few milliseconds. When implemented in MATLAB, the steering valve model simulation failed to produce output within the 10 μs time constraint. The vehicle subsystem, however, when simulated by itself, met its real-time constraint. We now apply the methodology discussed in Section 3.

5.1. Application of the methodology

Based on the criteria described in Section 3 for system-level analysis, we implemented the computation intensive part of the model



(a) Relative error for steering valve model



(b) Relative error for vehicle model

Fig. 13. Relative error in the output of steering valve and vehicle models.

(i.e. numerical integration method) in the hardware. The parallelism in the computations involved in the RK4 method made it an ideal candidate for FPGA implementation; the method to compute the position coordinates was simple enough to implement in software. The amount of communication between the steering valve model and the vehicle model logically led to the following two possible hardware/software partitioning schemes:

- Implement both of the computation-intensive models in hardware, since they can be efficiently parallelized and pipelined, and then compute the position coordinates in software, or
- Implement the steering valve model in hardware and the vehicle model and position coordinates in software, since the vehicle model can easily meet the real-time requirements as a software implementation.

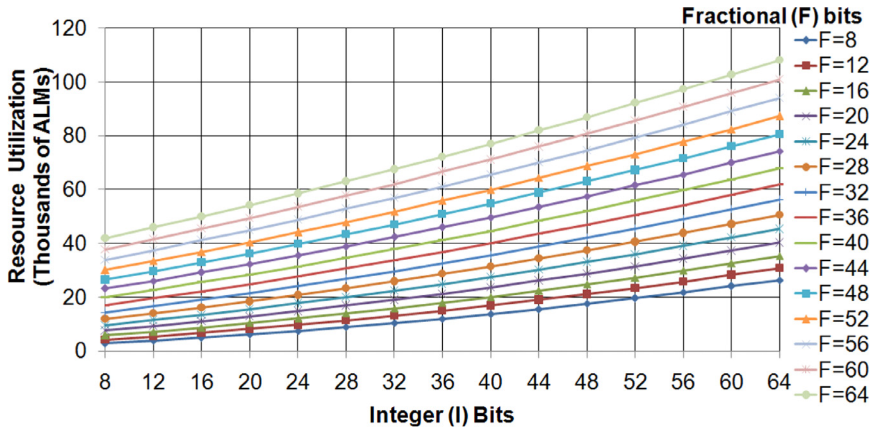
We discuss in detail the first approach and apply the methodology described in Fig. 5 to the vehicle system discussed above. The software partition and its implementation are discussed along with the implementation details of the selected hardware. We describe the design layout of both implementation strategies in Section 8.

Step 1: The inputs to Step 1 of our methodology were: the CPU-based simulator, a PRE of .01%, a RTC for the steering valve model of 10 μs, a RTC of 2 ms for the vehicle model, and 101, 760 $\times$ 2 ALMs of AR (Dual FPGA) [44]. To generate an fCPU-based simulator, we used a CPU-based simulator that implemented the dynamics of the vehicle system and found the functionalities that had an equivalent component in the hardware component library. In our vehicle system, the valve opening area unit used a linear-interpolation method, which was implemented using a hardware look-up curve component. The orifice flow rate unit used multiplication and square root functions. The multiplications were implemented using on-board hardware multipliers, and the square root function was implemented using a component from the library. The state-space solver for both the models used an RK4 component. After choosing the components, all the equivalent functionalities in the CPU-based simulator were replaced with the selected components from the software component library to obtain the fCPU-based design.

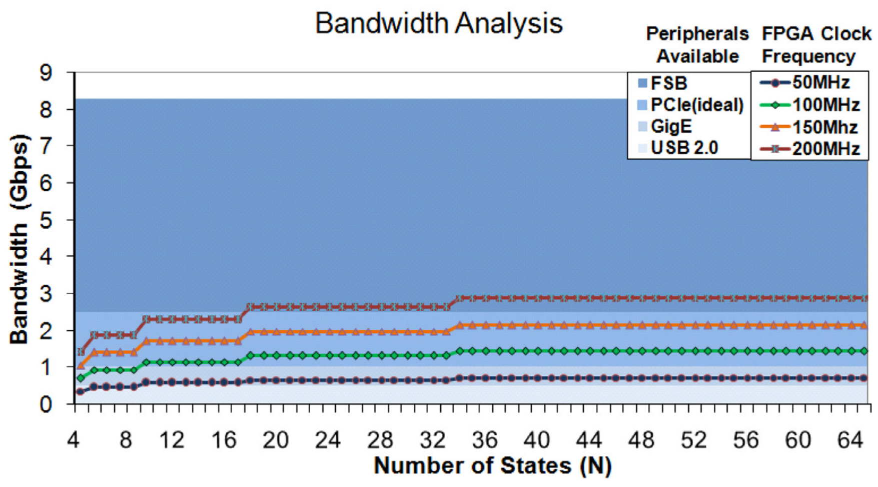
Step 2: For this work, we did not consider the variation in time with respect to the number of bits. We assumed that the time taken by the hardware components remained constant as the bit combination changed. To estimate the time taken for a completely parallelized design, we first enumerated the components in the order required to complete a single iteration of both the steering valve model and the vehicle model. We needed 4 look-up curve components, followed by a pipelined square root component for 6 orifices. The output of the square root component was used as input for the RK4 component for the steering valve model, whose output drove the RK4 component for the vehicle model. The output from all the look-up curve components was available after 5 cycles. The implementation of Eq. (5) for 6 orifices took 1 cycle to compute the two's complement of the negative number, 12 cycles to compute the square root, 1 cycle to again compute the two's complement of the result of the square root, and 1 cycle to compute the two multiplications. Thus, the output from all the orifices was available in 21 cycles. The RK4 components for the steering valve model and the vehicle model took 169 cycles and 113 cycles, respectively. An additional 7 cycles were consumed by a trigonometric function at the input of the RK4 component of the vehicle model to obtain the angular piston displacement. Thus, a single iteration of the steering valve model and the vehicle model took 315 cycles. The amount of time to compute a single iteration for a design with a 100 MHz clock was 3.15 μs; for a design with a 75 MHz clock, 4.2 μs; for a design with a 60 MHz clock, 5.25 μs. Each clock speed satisfied the 10 μs RTC.

Step 3: Ideally, the PRE is determined when the model is built, based upon the tolerable amount of error. For our work, the PRE was .01%. Given the maximum bit-width combination of $F = 64$ and $I = 64$, the plots for relative error in Fig. 13 show the PRE constraint was easily satisfied with this combination.

Steps 4, 5 and 6: We further reduced the bit-width combination by estimating the relative error for different combinations. We started with the maximum bit-width of 128, for which the relative error was close to zero. We first estimated the number of *I* bits for which the relative error remained within the given PRE value and then reduced the number of *F* bits until it satisfied the criteria. Fig. 13a shows the relative error in the output of the non-linear steering valve model for



(a) Hardware resource utilization for RK4 (vehicle model)



(b) Bandwidth analysis

Fig. 14. Estimates used for Step 7.

I ranging from 64 down to 48 with different values of F. As we reduced the number of F bits, the relative error increased until $F = 16$. However, for values of F less than 16, the data values were so small that they were converted to zeros due to insufficient bit-width. As a result, the relative error decreased to 100% for values of F less than 16. The number of F bits for which the relative error reached an error value close to the given range was thus estimated to be $I = 49$ and $F = 42$. Similarly, Fig. 13b shows the relative error in the output for the linear vehicle model for I ranging from 24 down to 8 with different values of F. As we reduced the number of I bits, the overlapping plots imply that the relative error remained constant; whereas it increased as we reduced the number of F bits. The number of F bits for which the relative error reached an error value close to the given range was thus estimated to be $I = 8$ and $F = 46$.

Step 7: The graphs in Figs. 6 and 14a were used to estimate the hardware RU for a completely parallelized design. The RU was computed in terms of Altera's ALMs. The steering valve model needed four look-up curve components, six square root components, one RK4 component, and a trigonometric function for a total RU of approximately 98K ALMs. The vehicle model needed one RK4 component for a RU of approximately 27K ALMs. The total RU was estimated to be 125K ALMs, which was less than the maximum AR of 203.52K ALMs. Since the PRE constraint was satisfied in Step 6, and the RU was less than the AR, the selection of hardware/software partitions was sufficient to generate an FPGA design. The parameters file and look-up tables for each component were automatically configured and generated by scripts to use data in fixed-point format with a bit-width of $M = I + F$ bits. A wrapper was used to connect the look-up curve, square root, and RK4 components together.

To check the design for functional correctness, the MATLAB version of the design was run for several iterations, and the output after each computation was converted to the equivalent hexadecimal fixed-point representation and logged to a file. Next, the hardware was simulated using Modelsim for the same number of iterations and the output compared to the results of the MATLAB simulation. The hardware simulation using the initial bit-width combination did not match the MATLAB output because there were an insufficient number of F bits needed to represent the small values generated during the computations. Therefore, it was necessary to increase the number of F bits and re-iterate through Step 7 until the hardware simulation sufficiently matched the MATLAB simulation. The final bit-width combination for the steering valve model was eventually selected with $I = 49$ and $F = 47$, and $I = 10$ and $F = 46$ for the vehicle model. The hardware RU then became 113K ALMs for the steering valve model and 30K ALMs for the vehicle model, for an approximate total of 143K ALMs, which still satisfied the RU constraint.

The next step was to determine the bandwidth requirement of the design generated in Step 7. An analysis of the bandwidth of available communication interfaces on Xilinx and Altera boards was conducted to find a suitable interface. Fig. 14b depicts the bandwidth requirement of the vehicle model (with only the RK4 component) versus the number of model states, as well as the

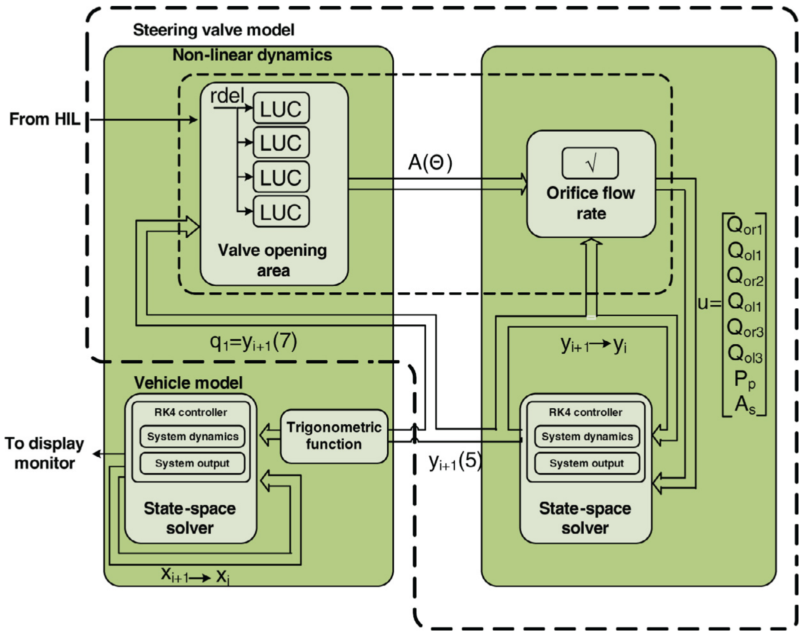


Fig. 15. Design layout of FPGA-based simulator for the vehicle system.

bandwidth capacities of various interfaces. The interfaces included in the analysis were: PCIe (2.5 Gb/s) [45], USB (0.48 Gb/s) [46], GigE (1 Gb/s) [47], and FSB (1.035 GB/s) [48]. The plot shows the required bandwidth increases as the order of the model increases. However, the bandwidth requirement is based upon the amount of time necessary to generate the output, so as the computation time of the model increases, the required bandwidth decreases. Since the complexity and order of the model increase the amount of time before the output is generated, the required bandwidth decreases. As a result, the bandwidth requirement of the system is expected to decrease when the steering valve and vehicle models are combined.

To obtain the required bandwidth for the steering valve and vehicle models running at 100 MHz, it was necessary to consider the maximum amount of data per unit time the FPGA sent and received from the CPU. The steering valve model required as inputs 128 bits for a reset signal, $128 \times N$ bits to define the initial state of the system, and 128 bits for the system input. Here, the number of states in the model was $N = 8$, so the maximum number of simultaneous data bits necessary for the steering valve model was 1280 bits. Since the system input was only received once every 20 ms, when the hardware finished the computation for a given system input, it idled until the next input arrived. The bandwidth from the CPU to the FPGA was estimated by $1280\text{ bits}/20\text{ ms} = 64\text{ Kb/s}$. The outputs from the vehicle model were: 8 states, each 128 bits wide, and a counter value, also 128 bits wide, for a total of 1152 bits every 20 ms sent from the FPGA to the CPU. So, the estimated return bandwidth was $1152\text{ bits}/20\text{ ms} = 57.6\text{ Kb/s}$. As Fig. 14b shows, every interface could provide adequate bandwidth for $N = 8$. Unfortunately, no single-board platform contained enough resources for the design, so a dual-board solution was chosen: the XtremeData platform with two Stratix 3SE260 boards provided 203.52K ALMs and a bandwidth on the order of 8.28 Gb/s (1.035 GB/s).

6. Design layout

From the CPU-based simulator, it is clear that the input of one unit depends upon the output of the previous unit. This sequential flow of information requires that a high-level architecture of the CPU-based simulator be maintained in the FPGA-based simulator. However, differences in the architecture of each unit change the way the dataflow is maintained in the FPGA. Fig. 15 shows how the components described in Section 4 are connected together to construct the data path.

For the first implementation strategy, the FPGA-based simulator of the vehicle system consisted of four major components: the valve opening area, the orifice flow rate, the state-space solver for the valve model, and the state-space solver for the vehicle model. Because of resource constraints on a single FPGA, we used both FPGAs (FPGA A and FPGA B) provided by the XtremeData platform [49]. The shaded blocks in Fig. 15 show the partitioning of the components. FPGA A implements the valve opening area of the steering valve model and the RK4 integrator of the vehicle model, whereas FPGA B implements the orifice flow rate and the RK4 integrator of the steering valve model. FPGA A is capable of exchanging data with the host and FPGA B, and FPGA B can exchange data only with FPGA A. Each wrapper that instantiates the components on the two FPGAs implements a finite state machine (FSM) that synchronizes the data-flow between the FPGAs and the host. Since the interface connecting the two FPGAs together, and FPGA A with the host, is 256 bits wide, the FSMs also pad the outgoing data with zeros as well as extract the required $I + F$ bits from the incoming data.

The wrapper on FPGA A scans the 32 most significant bits of the incoming data packet sent by the host. These bits categorize the first 128 bits of data in the packet as one of the following: simulator reset data, system state initialization data, and system input data, received in that order. When the system input is received, the valve opening area reads the $I + F$ bits and starts the computation. When the computation is finished, a “done” signal is asserted, marking the validity of the data, and the output vector is padded with zeros to make each value 128 bits wide. Each vector contains four 128-bit values, so two packets are required to transmit the entire vector across the bus. After the transmission completes, the component stalls until it receives new state information from the state-space solver of the steering valve model. Recall from Eq. (5) that $A(\theta) \times C_d$ and *sqrt* can be computed independently of each other. While the valve opening area is busy computing the former, the system input is also sent to the orifice flow rate to compute the latter. The orifice flow rate stalls after computing the *sqrt* and waits for a valid output from the valve opening area. The wrapper on FPGA B monitors the bus from FPGA A, waiting for a signal to indicate that valid data is available. The valid signal remains high for two clock cycles, since two 256-bit packets are sent. The output from the orifice flow rate is a vector of size eight, each element $I + F$ bits wide, and a ”done” signal to mark the validity of the data on the output bus. The orifice flow rate output becomes the system input for the steering valve state-space solver. The steering valve model starts computation when the ”done” signal is asserted. The output from the state-space solver is fed back to both the orifice flow rate and valve opening area. The orifice flow rate reads all the states of the system, whereas the valve opening area needs only the state variable q_1 (Fig. 3a).

The vehicle model starts execution only when the steering valve model has completed. However, because of the difference in their respective time steps, there is not a one-to-one relation between the number of times each one is executed. When the steering valve model completes S/h_{valve} iterations, the vehicle model then runs for $S/h_{vehicle}$ iterations. Thus, when the state-space solver for the steering valve completes its execution, the wrapper on FPGA B checks if S/h_{valve} iterations have completed. If not, then the FSM sends the state variable q_1 and x (piston displacement), used to compute the system input for the state-space solver for the vehicle model, to FPGA A and waits for the next valid output from the valve opening area. If so, then the FSM resets the counter to zero and waits for a new system input from the host to be routed through FPGA A. On FPGA A, a similar check is made to see if S/h_{valve} iterations have completed. If so, the trigonometric component reads x on the incoming interface from FPGA B, computes the angular displacement of the piston, and triggers the vehicle model to run for $S/h_{vehicle}$ iterations. At the same time, the counter for S/h_{valve} iterations is reset to 0, and the valve opening area component is set to wait for a new system input from the host.

The initial state of both the steering valve model and the vehicle model is assumed to be zero. Thus, when the first system input is received, the valve opening area does not read the incoming bus from FPGA B for the updated value of q_1. For the next S/h_{valve} iterations, the system input remains same, whereas state q_1 is updated after every iteration. The valve opening area component contains four look-up curve components that read the four different memory blocks where the data for each valve are stored. The length of the address to access the memory block is determined by the number of data values in the block, where each datum is $I + F$ bits long. The information about the address, maximum and minimum X values in each memory block, ΔX, and $1/\Delta X$ are stored in the parameters files when generating the VHDL design using the automated scripts.

The orifice flow rate component instantiates a square root component, the input of which is the difference between various states of the valve, determined by the state-space solver. The difference computation is pipelined, so a new input can be sent to the square root component every cycle. The output from the square-root component is latched until a valid output is received from the valve opening area.

The system input, u, for the steering valve model is a vector computed by the orifice flow rate component, given in Fig. 13b. The system input, u, for the vehicle model is a scalar quantity, computed by applying a sinusoidal inverse function to the piston displacement, x. For the vehicle system, the piston displacement is limited to the range $\pm$ 0.254, so instead of normalizing the input between 0 and 1, we saturate the input within the given range. A valid ”done” signal triggers the state-space solver for the vehicle model, which uses the output of the trigonometric component as system input.

For the second implementation strategy, we implemented the vehicle model in MATLAB and kept the steering valve model on the FPGA. The previous components were reused, but were put on a single FPGA to avoid the communication delay between the FPGAs. The state-space solver for the vehicle model reads the output of the steering valve model sent from the FPGA, and then runs for 20 ms.

7. Data flow and cycle estimate

The valve opening area uses the initial two cycles to compute A_m (Eq. (4)) and *rdel* (Eq. (3)). The division required for the computation of A_m is implemented with an inverse operation. Since I_g remains constant throughout the simulation, its inverse is saved in the parameters file when generating the VHDL design. *rdel* is computed in the second cycle by taking the two's complement of A_m and adding the result to A_s. The look-up curve takes five cycles to generate an output, a cycle to obtain the product with C_d, and a cycle to pipeline the result. Therefore, a total of 9 cycles are used to obtain the output from a single look-up curve component. Four look-up curve components for the four valves are instantiated in a pipelined manner, the output for which is available in 12 cycles. The output for each valve, $A(\Theta) \times C_d$, is saved in vector $\vec{A}(\Theta)$. An additional cycle is consumed by the wrapper to pad the data with zeros.

The orifice flow rate component reads $\vec{A}(\Theta)$, along with the present state of the system, y_i, and computes the flow rate using Eq. (7). Since the product $A(\Theta) \times C_d$ was computed by the valve opening area, the orifice flow rate handles two functions of Eq. (5): the square root and the multiplication of the square root with $\vec{A}(\Theta)$. In the first cycle, it computes the difference between the two pressure values $(p_i - p_f)$ by adding the two's complement of p_f to p_i. The absolute value of the difference is obtained by taking the two's complement only if the difference is negative (i.e the MSB is 1). To replicate the sign of the difference in the final output from this component, the sign bit is pipelined for use in a later stage. The square root of the difference is initiated in the second cycle and takes 12 cycles to finish. Thus, a single square root computation takes 14 cycles, and a pipelined architecture for six computations (for six flow rates) takes 19 cycles.

The valve opening area and orifice flow rate are executed in parallel. After receiving the new state information from the RK4 component, the orifice flow rate starts the computation immediately, stalling after 19 cycles, until it receives the latest value of $\vec{A}(\Theta)$ from the valve opening area unit. So, when the valve opening area starts the computation, the orifice flow rate is already waiting for input from the valve opening area unit. The time taken to complete one iteration of the steering valve includes 13 cycles for the valve opening area, followed by the 15 cycle delay associated with sending two 256-bit packets to FPGA B. The orifice flow rate takes 8 cycles to complete: 1 cycle to read the valve area, 6 cycles to compute six flow rate values, and 1 cycle to set up the input vector for the state-space solver. The state-space solver for the valve model takes 169 cycles, and another 13 cycles are used to send the two-state output to FPGA A. Thus, each iteration of the steering valve model is completed in 226 cycles. The trigonometric function on FPGA A takes 7 cycles to generate the system input. The state-space solver of the vehicle model reads this input and computes the new state in 113 cycles.

8. Implementation

The XtremeData XD2000i development system was chosen as the target platform for the FPGA-based simulator [49]. The system consisted of a development PC with a Xeon dual-processor system, running the Linux CentOS operating system, and an XD2000i FPGA in-socket accelerator that plugged directly into one of the CPU sockets. The XD2000i module featured three Stratix III EP3SE260 Altera FPGAs: one bridge FPGA and two application FPGAs (FPGA A and FPGA B), each with 254,400 logic elements (101,760 ALMs). The XD2000i module connected to the Xeon processor through a 1066 MT/s front-side bus (FSB) with a bandwidth of 8.5 GB/s. The bridge FPGA was dedicated to the FSB protocol between the CPU and application FPGAs and could not be modified. A 64-bit wide 200 MHz unidirectional bus connected the bridge FPGA to the two application FPGAs, and a 256-bit wide 100 MHz bus allowed data to pass between the application FPGAs.

A software controller to manage communication between the CPU and the application FPGAs ran on the Xeon processor. The functions to send and receive data from the application FPGAs were implemented

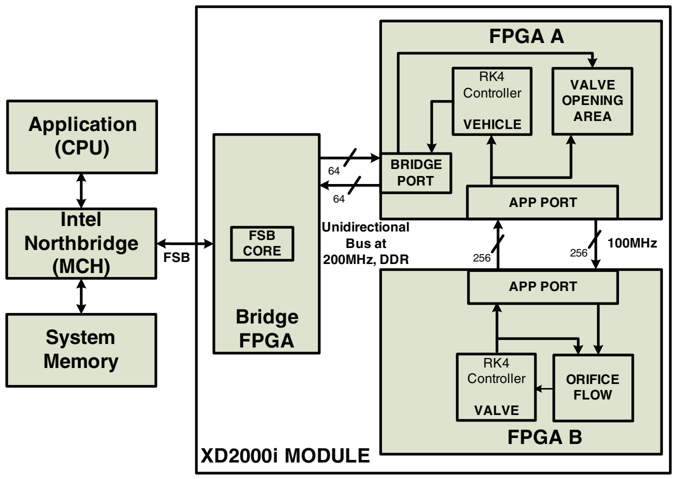


Fig. 16. XD2000i architecture and application mapping.

as blocking calls to ensure no new data was sent until the results from the previous transmission were returned. The software controller was aware of the number of bits a given transaction would send and the expected number of bits it would receive. Using this information, the controller created appropriately-sized send and receive buffers. However, any mismatch between the number of bits sent and received and the buffer sizes would cause both the controller and the hardware to stall.

The design partitioning across the two application FPGAs, shown in Fig. 16 and Table 5, was governed by two factors. First, the XtremeData platform required both input and output data exchanges to occur through the same application port. Hence, the valve opening area component, which received the system input, and the state-space solver of the vehicle model, which generated the simulation output, were implemented on the same FPGA. Second, care must be taken to appropriately divide the hardware components between the two application FPGAs in order to avoid incurring excessive communication overhead. For example, if the orifice flow rate component, which utilized the square root core, and the state-space solver for the steering valve model were implemented on separate FPGAs, the delay resulting from transmitting data for eight states every iteration would render the design useless.

The software controller sent the angle of the steering wheel to FPGA A every 20 ms, which allowed the hardware ample time to receive the new data, compute the new state of the vehicle, and send the results back to the CPU. Once the new state was computed, the hardware simulator stalled until a new input was received, and once the controller received the new state, it stalled for the remaining time.

9. Simulation and synthesis results

The FPGA-based vehicle simulator used a time step of 10^{-6} seconds for h_{valve} and a time step of 2^{-3} seconds for $h_{vehicle}$. The fastest clock at which the design could run and still meet the real-time constraints was 55 MHz (18.18 ns period), which was well below the maximum clock frequency of 100 MHz for the XD2000i platform.

Table 6
Modelsim times for each component.

Component	Time (μs)	
Steering valve model		4.1
Communication FPGA A → B		0.27
Communication FPGA B → A		0.234
Vehicle model		2.214
RK4	2.05	
Trigonometric	0.164	
Total		6.818

The Modelsim simulator was used to compare the time taken by a single iteration of the simulation on the FPGA with the time taken by a single iteration of the MATLAB-based simulator running on a 2.83 GHz Intel Core2 Quad processor. The MATLAB simulation completed one iteration of the steering valve and vehicle model in 13 μs, whereas the Modelsim simulation ran the same iteration in 6.818 μs. Of this 6.818 μs, 4.1 μs were consumed by the steering valve model, 0.27 μs (15 cycles) were used to send 512 bits of data from FPGA A to FPGA B, and another 0.234 μs (13 cycles) were used to send 256 bits of data from FPGA B to FPGA A. The total communication overhead between the FPGAs was 0.504 μs. Table 6 presents these results.

The vehicle model generated an output in 2.214 μs (123 cycles), 113 cycles of which were consumed by the RK4 component, seven cycles by the trigonometric function, 1 cycle to compute the input to the trigonometric function, and 1 cycle to complement the output of the trigonometric function. Therefore, the total time for one iteration of the vehicle simulation in Modelsim was $4.1 + 0.27 + 0.234 + 2.214 = 6.818$ μs, which resulted in a speedup of 2 × over the MATLAB implementation for the same iteration. The Modelsim simulation also showed that the final output from running the vehicle system for 20 ms (simulator time) would be generated after 9.21 ms (real time). In other words, the final state of the vehicle after 20 ms would be known after just 9.21 ms of real time, which is also a speedup of 2 × . The time required to send the data over the FSB was negligible when compared to the computation time of the model.

A timer was used to measure the execution time of the actual simulation. The timer starts when the software controller sends the system input to the FPGA and stops after the simulation results have been received. The measured time was 10 ms, verifying the projected 2 × speedup over 20 ms. The 10 ms includes the time required to send the system input to the FPGA, run the simulation on the FPGA, and receive the simulation results from the FPGA.

For the second implementation, we were able to implement the steering valve model on a single FPGA and run the design at 62.5 MHz. It took 7 ms to simulate 20 ms, excluding a 1 ms communication delay and 12 ms software stall. The simulation results were then sent to the vehicle model on the FPGA, which ran the vehicle model for 20 ms and computed the position coordinates to send to the VR display monitor. Since this second implementation used 3 different

Table 5
Resource utilization of each component in percentage of ALMs and communication data overhead.

Resource usage	Component	%	FPGA	Total
	RK4 (vehicle)	29	A	45%
	Valve opening area	11	A	
	Trig. function	5	A	
	RK4 (valve)	38	B	88%
	Orifice flow	50	B	
Communication overhead		Bitwidth	Bandwidth (Kb/s)	
	Input data	1280	64	
	Output data	1152	57.6	

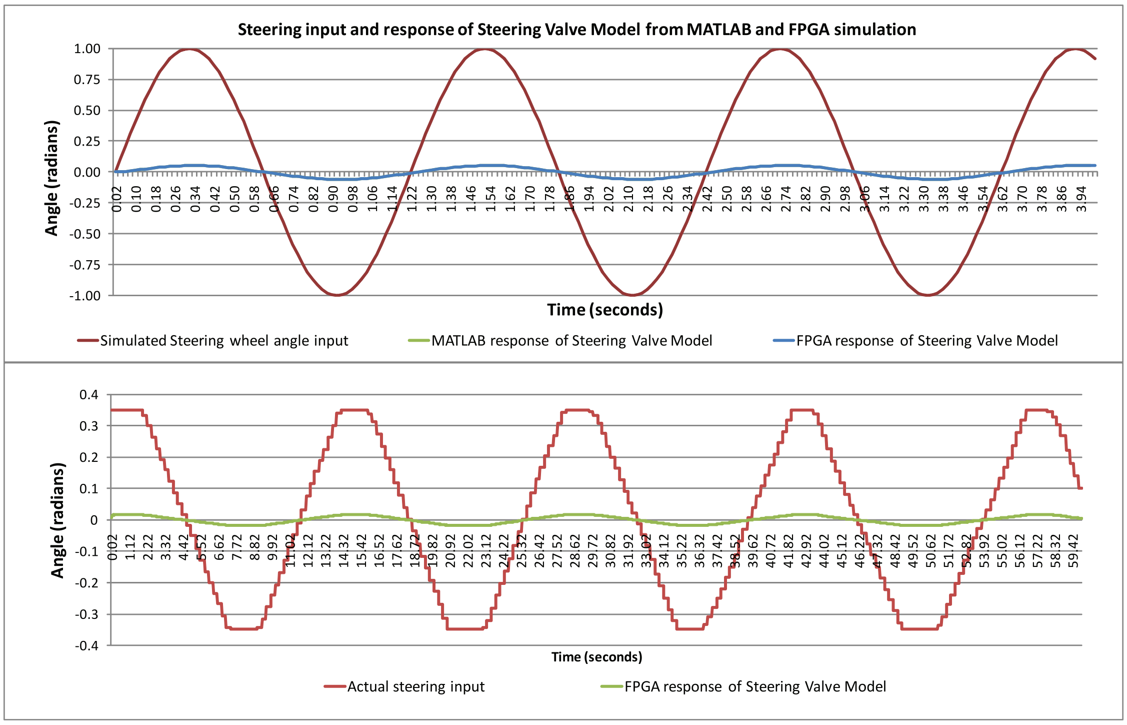


Fig. 17. Response of the steering valve model to simulated and actual steering input.

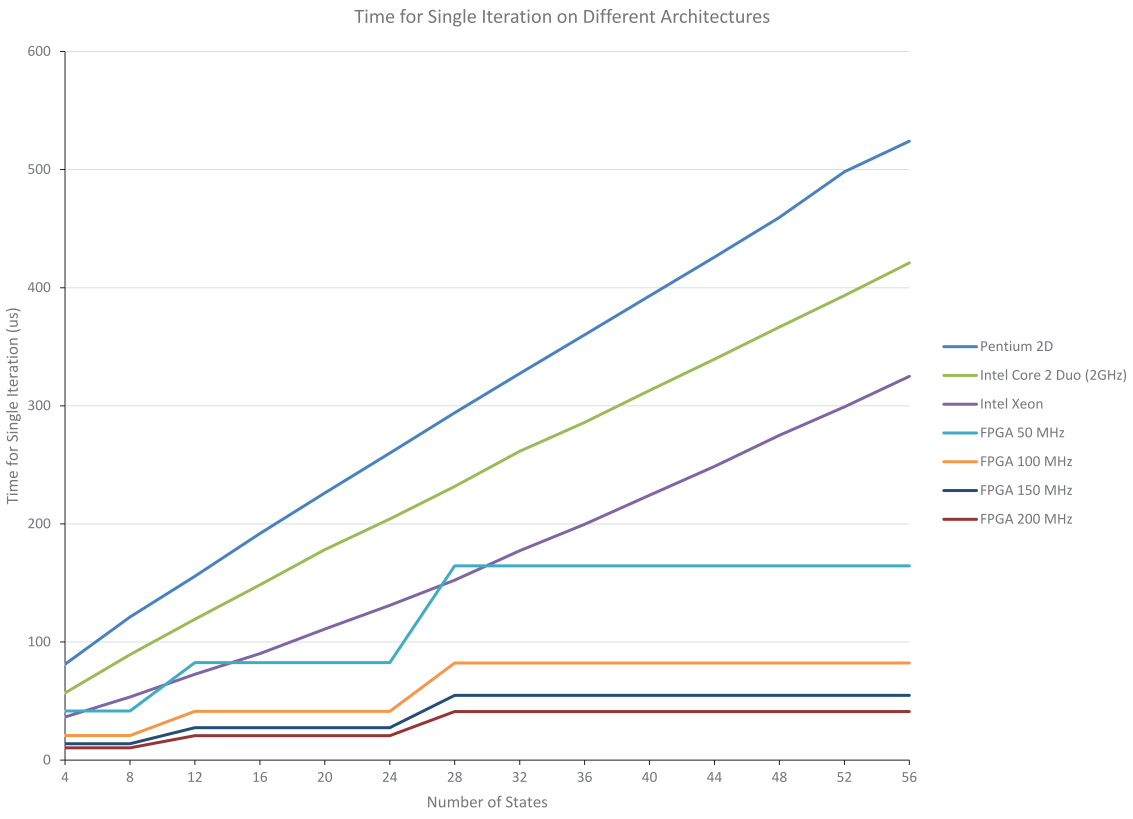


Fig. 18. The time for a single iteration of the simulation on different architectures, in microseconds.

platforms to drive the simulation, the simulation lagged slightly, due to the socket communication delay, which was 16 μs on average. If the computations were too slow, then, over a period of time, the accumulated error would cause the vehicle movement in response to the steering input to become sluggish. To show that the steering valve model on the FPGA was generating the correct results to pass to the vehicle model, the piston displacement angles computed by the steering valve model in response to different steering input sources were compared. The first source was a simulated sinusoid and the second was the actual steering wheel. Fig. 17 shows the results of this comparison. The piston displacement angle for the same set of input from the MATLAB version of the model was also plotted, and shows the MATLAB output overlaps with the FPGA output, further verifying the correctness of the FPGA steering valve model implementation.

The time taken for a single iteration on different architectures is shown in Fig. 18. The plot shows that of the three Intel processors, the Xeon performs the best as the number states increases. However, all but the slowest-clocked FPGA implementation outperforms the Xeon at even the smallest number of states. The performance of the three CPU architectures scales with the number of states approximately linearly, whereas the FPGA architectures scale in a step-wise fashion. This means that the FPGA solution scales better as the number of states in the system increases.

10. Conclusion

In this paper, we introduced a method to improve the simulation time of vehicle systems by using hardware implementations of the necessary mathematical models. We presented a methodology based upon estimating the resource usage of the hardware design early in the design process in order to make intelligent decisions about the implementation strategy. Although the methodology is suitable for different sizes of models, this paper showed it applied to an 8th order steering valve and vehicle model. The system was successfully implemented on a high-performance reconfigurable computing platform and was able to run the simulation twice as fast as the software-only simulator. As a side-effect of this research, a library of hardware components was designed and can now be reused for other models. While this paper focused on implementing a model across two FPGAs, the methodology can be formalized to consider any number of hardware platforms to obtain the best hardware/software partitions. Future work will focus on improving the existing partitioning algorithms presented in this work, as well as developing algorithms to better distribute the components of a model across different hardware architectures while still remaining integrated with software.

References

[1] S. Zheng, S. Zheng, J. He, J. Han, An optimized distributed real-time simulation framework for high fidelity flight simulator research, in: Proceedings of International Conference on Information and Automation (ICIA), 2009, pp. 1597–1601, doi:10.1109/ICINFA.2009.5205172.

[2] P. Le-Huy, S. Guerette, L. Dessaint, H. Le-Huy, Dual-step real-time simulation of power electronic converters using an FPGA, in: Proceedings of IEEE International Symposium on Industrial Electronics, 2006, pp. 1548–1553.

[3] X. Xiaobo, Z. Kangfeng, Y. Yixian, X. Guoai, A model for real-time simulation of large-scale networks based on network processor, in: Proceedings of the Broadband Network Multimedia Technology (IC-BNMT), 2009, pp. 237–241, doi:10.1109/ICBNMT.2009.5348486.

[4] M. Tavernini, B. Niemoeller, P. Krein, Real-time low-level simulation of hybrid vehicle systems for hardware-in-the-loop applications, in: Proceedings of Vehicle Power and Propulsion Conference (VPPC), 2009, pp. 890–895, doi:10.1109/VPPC.2009.5289753.

[5] J. Maroto, E. Delso, J. Felez, J. Cabanellas, Real-time traffic simulation with a microscopic model, IEEE Trans. Intell. Transp. Syst. 7 (4) (2006) 513–527, doi:10.1109/TITS.2006.883937.

[6] M. Lerotic, S.-L. Lee, J. Keegan, G.-Z. Yang, Image constrained finite element modelling for real-time surgical simulation and guidance, in: Proceedings of Biomedical Imaging: From Nano to Macro, 2009, pp. 1063–1066, doi:10.1109/ISBI.2009.5193239.

[7] M. Karkee, B.L. Steward, Open and closed loop system characteristics of a tractor and an implement dynamic model, in: Proceedings of the Annual Meeting of the American Society of Agricultural and Biological Engineers (ASABE), 2008.

[8] B.T. Kulakowski, J.F. Gardner, J.L. Shearer, Dynamic Modeling and Control of Engineering Systems, 3rd, Cambridge University Press, 2007.

[9] H. Zhou, G. Tong, C. Liu, GPES: a preemptive execution system for GPGPU computing, in: IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), 2015, pp. 87–97, doi:10.1109/RTAS.2015.7108420.

[10] C. Liu, J. Anderson, Suspension-aware analysis for hard real-time multiprocessor scheduling, in: Proceedings of the Euromicro Conference on Real-Time Systems (ECRTS), 2013, pp. 271–281, doi:10.1109/ECRTS.2013.36.

[11] M. Karkee, Real-time simulation and visualization architecture with field programmable gate array (FPGA), in: Proceedings of the ASME World Conference on Innovative Virtual Reality (WINVR), 2010.

[12] S. Hauck, The roles of FPGAs in reprogrammable systems, Proc. IEEE 86 (4) (1998) 615–638, doi:10.1109/5.663540.

[13] S. Note, P. van Lierop, J. van Ginderdeuren, Rapid prototyping of DSP systems: requirements and solutions, in: Proceedings of IEEE International Workshop on Rapid System Prototyping, 1995, pp. 88–96, doi:10.1109/IWRSP.1995.518576.

[14] H. Schmit, D. Thomas, Hidden markov modeling and fuzzy controllers in FPGAs, in: Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, 1995, pp. 214–221, doi:10.1109/FPGA.1995.477428.

[15] M. Serra, K. Kent, Using FPGAs to solve the hamiltonian cycle problem, in: Proceedings of the International Symposium on Circuits and Systems (ISCAS), 2003, pp. 228–231.

[16] J. Durbano, F. Ortiz, FPGA-based acceleration of the 3d finite-difference time-domain method, in: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 156–163, doi:10.1109/FCCM.2004.37.

[17] G. Morris, V. Prasanna, An FPGA-based floating-point Jacobi iterative solver, in: Proceedings of International Symposium on Parallel Architectures, Algorithms and Networks (ISPAN), 2005, p. 8, doi:10.1109/ISPAN.2005.18.

[18] B. Pandita, S. Roy, Design and implementation of a viterbi decoder using FPGAs, in: Proceedings of the International Conference on VLSI Design, 1999, pp. 611–614, doi:10.1109/ICVD.1999.745223.

[19] N. Azizi, I. Kuon, A. Egier, A. Darabiha, P. Chow, Reconfigurable molecular dynamics simulator, in: Proceedings of Field-Programmable Custom Computing Machines (FCCM), 2004, pp. 197–206, doi:10.1109/FCCM.2004.48.

[20] J.-H. Kim, C.-S. Park, S.-G. Kim, J.-H. Kim, Improved interpolation and system integration for FPGA-based molecular dynamics simulations, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 2006, pp. 21–28.

[21] R. Scrofano, M.B. Gokhale, F. Trouw, V.K. Prasanna, A hardware/software approach to molecular dynamics on reconfigurable computers, in: Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2006, pp. 23–34.

[22] Y. Gu, T.V. Martin, C. Herbordt, FPGA-based multigrid computation for molecular dynamics simulations, in: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2007, pp. 117–126.

[23] D. Lavenier, Speeding up genome computations with a systolic accelerator, 1998, http://www.irisa.fr/symbiose/lavenier/Publications/Lav98ja.pdf (accessed June 2015).

[24] S. Dydel, Large scale protein sequence alignment using FPGA reprogrammable logic devices, in: Proceedings of International Conference on Field Programmable Logic and Application (FPL), 2004, pp. 23–32.

[25] A. Boukerche, J. Correa, A. de Melo, R. Jacobi, A. Rocha, An FPGA-based accelerator for multiple biological sequence alignment with DIALIGN, in: High Performance Computing (HiPC), in: Lecture Notes in Computer Science, 4873, Springer Berlin / Heidelberg, 2007, pp. 71–82.

[26] G. Zhang, P. Leong, C. Ho, K. Tsoi, C. Cheung, D.-U. Lee, R. Cheung, W. Luk, Reconfigurable acceleration for Monte Carlo based financial simulation, in: Proceedings of International Conference on Field-Programmable Technology (FPL), 2005, pp. 215–222, doi:10.1109/FPT.2005.1568549.

[27] D.B. Thomas, J.A. Bower, W. Luk, Automatic generation and optimisation of reconfigurable financial monte-carlo simulations, in: Proceedings of International Conference on Application-specific Systems, Architectures and Processors (ASAP), 2007, pp. 168–173.

[28] N. Woods, T. VanCourt, FPGA acceleration of quasi-Monte Carlo in finance, in: Proceedings of International Conference on Field Programmable Logic and Applications (FPL), 2008, pp. 335–340, doi:10.1109/FPL.2008.4629954.

[29] A. Irturk, B. Benson, N. Laptev, R. Kastner, FPGA acceleration of mean variance framework for optimal asset allocation, in: Proceedings of Workshop on High Performance Computational Finance (WHPCF), 2008, pp. 1–8.

[30] D.B. Thomas, W. Luk, Sampling from the multivariate Gaussian distribution using reconfigurable hardware, in: Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM), 2007, pp. 3–12.

[31] C. Dufour, S. Abourida, J. Belanger, Real-time simulation of permanent magnet motor drive on FPGA chip for high-bandwidth controller tests and validation, in: Proceedings of IEEE International Symposium on Industrial Electronics, 2006, pp. 2591–2596, doi:10.1109/ISIE.2006.295980.

[32] C. Dufour, J. Belanger, S. Abourida, V. Lapointe, FPGA-based real-time simulation of finite-element analysis permanent magnet synchronous machine drives, in: Proceedings of Power Electronics Specialists Conference (PESC), 2007, pp. 909–915, doi:10.1109/PESC.2007.4342109.

[33] Y. Chen, V. Dinavahi, FPGA-based real-time EMTP, IEEE Trans. Power Deliv. 24 (2) (2009) 892–902, doi:10.1109/TPWRD.2008.923392.

[34] Y. Zhou, T. Mei, S. Freear, Field programmable gate array implementation of wheel-rail contact laws, Control Theory Appl., IET 4 (2) (2010) 303–313, doi:10.1049/iet-cta.2008.0601.

[35] R. Yates, Fixed-point arithmetic: an introduction, 2007, http://www.digitalsignallabs.com/fp.pdf (accessed June 2015).

[36] A. Percey, Advantages of the Virtex-5 FPGA 6-Input LUT Architecture, 2007, http://www.xilinx.com/support/documentation/white_papers/wp284.pdf (accessed June 2015).

[37] Xilinx, LogiCORE IP CORDIC (DS858), 2011, V5.0.

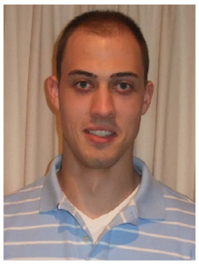
[38] Altera, Floating-Point IP Cores User Guide (UG-01058), 2014.

[39] P. Montuschi, P. Mezzalama, Survey of square rooting algorithms, IEEE Proc. Comput. Digital Tech. 137 (1) (1990) 31–40.

[40] P. Soderquist, M. Leeser, An area/performance comparison of subtractive and multiplicative divide/square root implementations, in: Proceedings of the Symposium on Computer Arithmetic, 1995, pp. 132–139, doi:10.1109/ARITH.1995.465366.

[41] G. Cappuccino, P. Corsonello, G. Cocorullo, High performance VLSI modules for division and square root, Microprocess. Microsyst. 22 (5) (1998) 239–246, doi:10.1016/S0141-9331(98)00082-9.
[42] P. Soderquist, M. Leeser, Division and square root: choosing the right implementation, IEEE Micro 17 (4) (1997) 56–66, doi:10.1109/40.612224.
[43] Chapra, Canale, Numerical Methods for Engineers, 5th, Tata McGraw-Hill, 2006.
[44] M. Karkee, Modeling, identification and analysis of tractor and single axle towed implement system, Iowa State University, 2009 (Ph.D. thesis). Paper 10875.
[45] PCI Express Base Specification, PCI-SIG, 2007. Rev. 2.0.
[46] Compaq, Hewlett-Packard, Intel, Lucent, Microsoft, NEC, Philips, Universal Serial Bus Specification, 2000. Rev. 2.0.
[47] IEEE Standard for Ethernet, IEEE, 2008.
[48] XtremeData, Xd2000i™development system, http://www.altima.co.jp/products/xtremeData/download/XD2000i_PF_WEB.pdf (accessed June 2015).
[49] XtremeData, Xd2000i™development system, (accessed June 2015).

Madhu Monga received her B.S. degree in Electronics and Communication from Kurukshetra University in 2003, and her M.S. degree in Electrical and Computer Engineering from Iowa State University in 2010. She was a research assistant in the Reconfigurable Computing Lab until 2010 and worked on accelerating real-time simulation of dynamic systems. She is currently employed at Algo-Logic. Her research interests include high performance reconfigurable computing and embedded systems.


Daniel Roggow received his B.S. degree in Computer Engineering from Iowa State University, Ames, IA, in 2014. He is currently a research assistant working on his Ph.D. in the Reconfigurable Computing Laboratory at Iowa State. His research interests include reconfigurable computing and embedded systems.

Manoj Karkee is an assistant professor in the Biological Systems Engineering Department. Dr. Karkee was born in Nepal where he received his undergraduate degree in Computer Engineering. He went to Asian Institute of Technology, Bangkok, Thailand in 2003 for his Master's Degree in Remote Sensing and GIS. He received his Ph.D. in Agricultural Engineering and Human Computer Interaction from Iowa State University in 2009 and joined WSU in 2010. Dr. Karkee has established a strong research program in agricultural automation and mechanization area with particular emphasis on machine vision systems. Some of his sponsored projects include crop load estimation, fruit tree pruning, plant stress monitoring, and solid set canopy delivery. He has been publishing actively in journals such as 'Computers and Electronics in Agriculture', and 'The Transactions of ASABE' and has been presenting frequently in national and international conferences.

Song Sun received his B.S. degree in Computer Science and Engineering from the Beijing Information Technology Institute in 1998, his M.S. degree from the University of Science and Technology of China in 2001, and his Ph.D. degree in Electrical and Computer Engineering from Iowa State University, Ames, IA, in 2011. He currently is working at the Symantec Corporation. His research interests include high performance reconfigurable computing and embedded systems.

Lakshmi Kiran Tondehal received her B.S in Electronics and Instrumentation from the Birla Institute of Technology and Science (BITS) Pilani, Goa campus in 2009, and her M.S. in Electrical and Computer Engineering from Iowa State University, Ames IA in 2011. Her research interests are in embedded systems, reconfigurable computing and VlSI design. She is currently a Memory Product Engineer at Micron Technology, Boise ID.

Brian Steward is Professor of Agricultural and Biosystems Engineering at Iowa State University. Dr. Steward teaches in the areas of dynamic systems modeling and simulation, fluid power engineering and technology, computational intelligence, and sustainable engineering. His research focuses on the areas of dynamic systems and sensing for agriculture and biosystems, precision agriculture, and field automation and mechatronics at the interface of machine and environment. He provides leadership to international programs within the ABE Department including coordination of student exchange programs with Federal University of Viçosa, Viçosa, Minas Gerais, Brazil, Federal University of Campina Grande, Paraiba, Brazil and study abroad trips to Brazil. He is a member of the American Society of Agricultural and Biological Engineers in which he been an associate editor for seven years and currently chairs the Resource Editorial Board. He has a total of 177 technical publications including 35 refereed journal articles and 44 invited talks. He has received more than $4.8 million in grants and contracts as a principal investigator or co-principal investigator. He received his BS and MS in Electrical Engineering from South Dakota State University, and his Ph.D. in Agricultural Engineering from the University of Illinois at Urbana-Champaign.

Atul Kelkar is a Professor of Mechanical Engineering at Iowa State University and prominent scholar in the area of Dynamic Systems and Control. Dr. Kelkar received his Ph.D. degree in Mechanical Engineering from Old Dominion University, Norfolk, Virginia, in 1993 while working as a Research Scientist at NASA Langley Research Center, Hampton, VA. Dr. Kelkar joined Iowa State University in 2001 and currently holds the position of Professor in Mechanical Engineering Department.

Joseph Zambreno (M '02) received his B.S. degree (Hons.) in Computer Engineering in 2001, his M.S. degree in Electrical and Computer Engineering in 2002, and his Ph.D. degree in Electrical and Computer Engineering from Northwestern University, Evanston, IL, in 2006. Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering at Iowa State University, Ames, where he has been since 2006. His research interests include computer architecture, compilers, embedded systems, and hardware/software co-design, with a focus on run-time reconfigurable architectures and compiler techniques for software protection. Dr. Zambreno was a recipient of a National Science Foundation Graduate Research Fellowship, a Northwestern University Graduate School Fellowship, a Walter P. Murphy Fellowship, and the Electrical Engineering and Computer Science Department Best Dissertation Award for his Ph.D. dissertation “Compiler and Architectural Approaches to Software Protection and Security”.