2010

Real-time simulation of dynamic vehicle models using high performance reconfigurable computing platforms

Madhu Monga
Iowa State University

Real-time simulation of dynamic vehicle models using high performance reconfigurable computing platforms

by

Madhu Monga

A thesis submitted to the graduate faculty

in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

Major: Computer Engineering

Program of Study Committee:
Joseph Zambreno, Major Professor
Brian L. Steward
Atul G. Kelkar
Dionysios C. Aliprantis

Iowa State University

Ames, Iowa

2010

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

DEDICATION

To,

My parents

and

In loving memory of my grandparents.

ACKNOWLEDGEMENTS

I would like to thank Dr. Joseph Zambreno for his continuous support and feedback throughout the entire duration of this course work. I appreciate his patience and effort in explaining things which helped create a very comfortable and encouraging work environment. He always provided a different perspective to the various situations faced and helped me make this thesis better. Working with him has helped me immensely, both on personal and professional fronts.

I would also like to thank Dr. Brian Steward and Dr. Atul Kelkar for their valuable suggestions in improving the implementations, and Dr. Dionysios Aliprantis for providing a different outlook to this research.

This section would be incomplete without extending my thanks to Dr. Phillip Jones, who has been a great mentor, all throughout the graduate school. I really appreciate him for his time and effort spent in order to help me understand different hardware platforms. A note of thanks to Dr. Manoj Karkee, who took his time out to explain me different aspects of the vehicle system dynamics and to my team members and lab mates.

Last but not the least, it is all because of immense support and love from my family and my friends, whose love and encouragement have always given me strength to keep moving ahead.

ABSTRACT

A software-based approach for Real-Time Simulation (RTS) may have difficulties in meeting real-time constraints for complex models. In this thesis, we present a methodology for the design and implementation of RTS algorithms, based on the use of Field-Programmable Gate Array (FPGA) technology to improve the response time of these models. Our methodology utilizes traditional Hardware/Software co-design approaches to generate a heterogeneous architecture for an FPGA-based simulator. We have optimized the hardware design such that it efficiently utilizes the parallel nature of FPGAs and pipelines the independent operations. Further enhancement is obtained through the use of custom custom accelerators for common non-linear functions. Since the systems we examine have relatively low response time requirements, our approach greatly simplifies the software components by porting the computationally complex regions to hardware. 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.

CHAPTER 1. Introduction

Real-time simulation (RTS) is comprised of a set of mathematical techniques 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 [54], power systems [22], networking [50], automotive [44], traffic management [26], and medicine [24]. For example, the rapid growth in Internet technology has resulted in very complex and widespread networks which are not appropriate for testing new network protocols and topologies. RTS of such large-scale networks provide researchers with a high-fidelity and scalable testbed [50]. In biomedical engineering, tissue deformation due to respiratory motion is an important research area since it can be used in treatment of several diseases. RTS of respiratory motion for each patient is thus useful for studying the deformities of the tissue over a period of time [24]. In the development of off-road vehicles, such as in military and agricultural applications, in order to reduce the development cost and rework time it is essential to determine the real-time behavior of each physical component involved in the system.

These dynamical systems are mathematically modelled by determining functions that represent the behavior of the forces acting upon its components. To simulate them and measure their mathematical state at different instances in time, these models are integrated using numerical integration algorithms such as Runge-Kutta (RK1, RK2, RK4), Adams-Bashforth, Adams-Moulton etc. These algorithms employ either fixed or dynamically changing time steps. The response generated by the simulation after each time step is considered useful for real-time analysis only if the computation time for a single iteration remains below or equal to the actual time being simulated. Another common feature is a user-interactive environment which ensures that users are able to modify the system input in real-time, thus simultaneously observing its effect in order to make changes to improve the behavior of the system. Virtual Reality (VR) technology has been employed in setting up such virtual environments, where simulation systems consist of a user control, simulation model, and display monitor. User-control generates or sends real-time system input for the simulation model and the display monitor displays its effect on the physical system being modeled.

Platforms for implementation of simulation models can be traced back to the 1950s when engineers used analog or digital computers [35] or a combination of both [23], [36] to simulate systems in real-time. The programming languages used back then were mainly assembly-level or FORTRAN. 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. Over the past few years, considerable progress has been made in RTS of power systems [48], [8], [31] and vehicle systems [1], [34], [37], [7] using digital computers. However, the general-purpose software processor (i.e. the 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 execution of these computations thus fails to cope with the real-time constraints which further restrict the usefulness of RTS in a VR environment.

In this thesis, we focus on acceleration of real-time Hardware-in-the-Loop (HIL) simulation of vehicle systems by implementing the numerical integration algorithms using reconfigurable hardware, i.e. Field-Programmable Gate Arrays (FPGAs) [16]. FPGAs make an ideal choice for accelerating this class of algorithms since it provides a platform to parallelize the independent computations. Custom pipelined architecture provides an opportunity to improve the throughput, though at the expense of an initial latency. More work per clock cycle thus results in a tremendous improvement in the computation time. We aim to improve end-to-end computation time for vehicle system simulation. This is triggered when a system input is sent from the user-control to the simulation model and ends when hardware sends the simulation

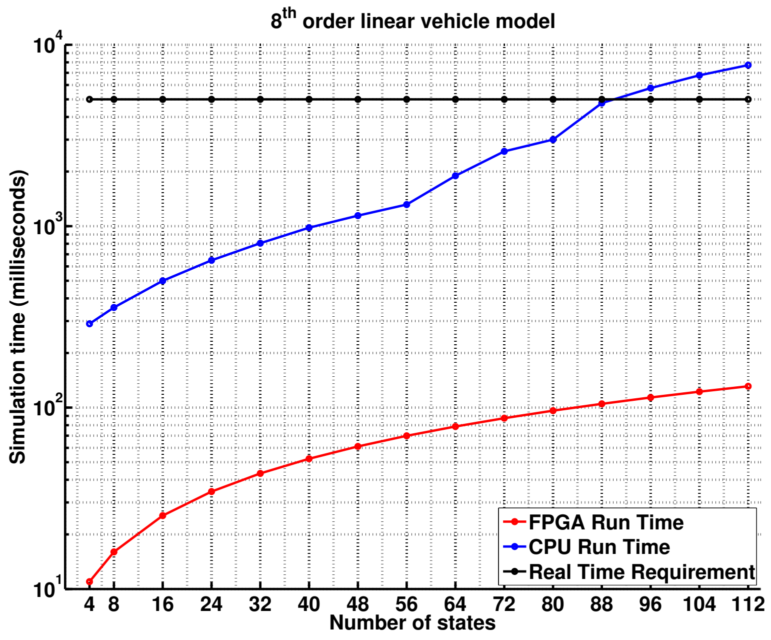


Figure 1.1 FPGA vs CPU vs real-time simulation with different number of states

results back to the display monitor.

The dynamic behavior of a vehicle system is determined by the various forces that act on its subsystems, such as steering, acceleration, braking, chassis, and tires. These forces in turn affect the steering direction, stability, velocity, suspension, and displacement of the vehicle either in a linear or non-linear fashion [25]. RTS of a vehicle system comprising of these subsystems thus ensures that dynamic behavior of the actual vehicle remains within the safe range of operation. The vehicle system chosen for hardware implementation in this thesis consists of a steering subsystem, which defines the directional dynamics of the vehicle in response to the steering input, affected by the linear and non-linear forces. To simulate the system in real-time with a time step of 10μs for non-linear forces and 2ms for linear forces, we first used a CPU-based (MATLAB) simulator. In order to meet the real-time constraints, it was necessary that the time required for sending the system input from user-control, running the simulation and sending the output to the display monitor, should in total be less than the selected time step of 10μs. However, it was observed that the time taken to run the simulation by itself was 13μs. Figure

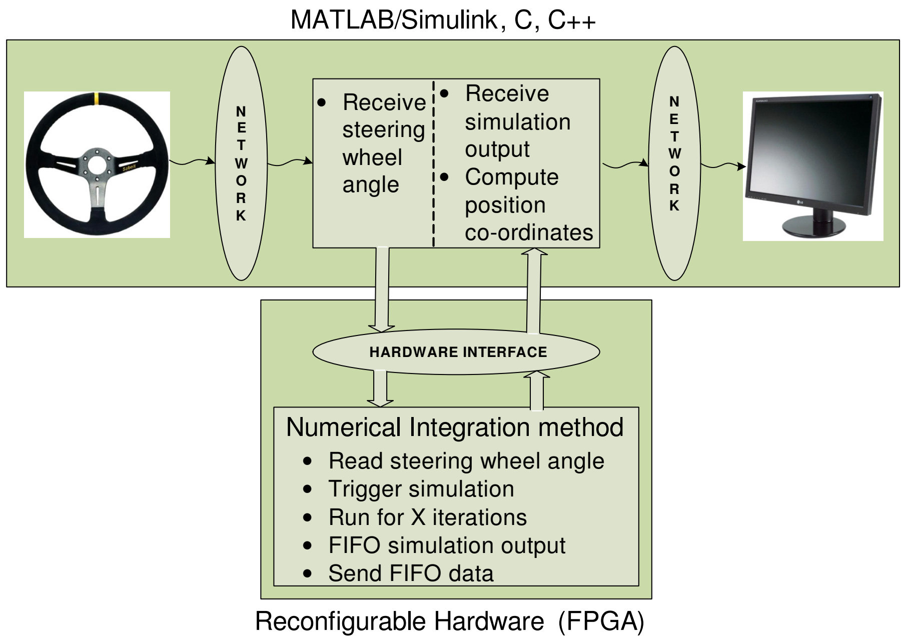

MATLAB/Simulink, C, C++
NETWORK
Receive steering wheel angle
Receive simulation output
Compute position co-ordinates
NETWORK
HARDWARE INTERFACE
Numerical Integration method
Read steering wheel angle
Trigger simulation
Run for X iterations
FIFO simulation output
Send FIFO data
Reconfigurable Hardware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

Figure 1.2 Simulation architecture

1.1 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, acted upon by linear forces for a simulation period of five seconds on an FPGA running at 55.55MHz and MATLAB on an Intel Core 2 Quad CPU running at 2.83GHz using the RK4 integration method. 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 being modelled for the system is greater than 88. On the other hand, for the FPGA-based simulator, the computation time remains well below the real-time constraint. We can intuitively say that a more complex system with additional subsystems and forces will further add to the time taken per iteration and result in violation of constraint even with a lesser number of states.

The steering subsystem includes steering valve dynamics acted upon by the non-linear forces and the lateral dynamics of the vehicle acted upon by the linear forces. The former simulates the delay between the rotation of the steering wheel and the actual movement of the wheel and the

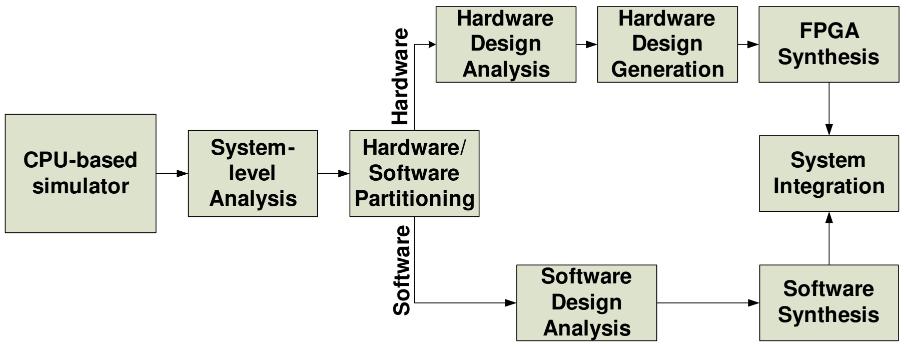


Figure 1.3 Hardware/Software co-design approach for Vehicle System Simulation

latter simulates the movement of the vehicle. The simulation model of the vehicle system thus consists of a non-linear steering valve model to simulate the valve dynamics and a linear vehicle model to simulate the vehicle dynamics. For the hardware implementation, we first simulated the vehicle model with the complete simulation set-up. It consisted of HIL (the actual steering wheel), which sent the steering wheel angle to the vehicle model, and a display monitor that rendered the movement of the vehicle, based on the position coordinates, computed using the output of the simulation. We then added the steering valve model which was connected to the HIL and received the steering wheel angle instead of the vehicle model. Its output was hence used to drive the vehicle model. For this thesis we focus on its hardware implementation. The simulation architecture is as shown in Figure 1.2 and includes the user control modeled by the HIL, and the simulation model, developed using Very-High-Speed Integrated Circuit (VHSIC) Hardware Descriptive Language (VHDL), which runs on the hardware, as well as a display monitor that displays the movement of the vehicle in response to the steering wheel angle input.

We propose a Hardware/Software co-design approach to accelerate the RTS using a heterogeneous parallel architecture. Figure 1.3 provides an overview of our approach. Using this approach we claim the following contributions to the state-of-art in simulation of vehicle system dynamics which otherwise fail to meet the real-time constraints using software (CPU-based) simulator:

- A co-design approach for RTS by partitioning the tasks between hardware and software platform.

- A methodology based on heuristic approach to generate an FPGA-based simulator. The approach uses 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 Hardware/Software based simulator.

Using this approach, we first implemented the RTS for linear vehicle model with a time step of 2ms. Even though this model when simulated on MATLAB meets the real-time constraints, to show proof-of-concept and feasibility of real-time simulation using FPGA, we first implemented this simple version. In addition to achieving a speedup of 17× for a simulation time of 20ms, we were successfully able to meet the real-time constraints with HIL and display monitor in the setup. A more complex steering valve model with a very smaller time step of 10μs was then added which imposed a tighter constraint on the real-time requirement. A speedup of 3× was achieved for the steering valve model.

The rest of this thesis is organized as follows. In Chapter II we present an overview of the dynamic vehicle system simulation. In Chapter III, we explain our co-design approach for partitioning, followed by our methodology for generating an FPGA-based simulator of the vehicle system. Before we apply this approach to an example system we present the hardware implementation details for some commonly occurring non-linear components in Chapter IV. We apply our methodology to an example vehicle system whose behavior is controlled by non-linear steering valve and linear vehicle dynamics in Chapter V. We describe its hardware architecture followed by implementation details and results on the XtremeData platform across dual FPGAs. This is followed by a brief discussion about the related work in the field of FPGA-based acceleration methods in Chapter VI. We conclude the thesis with summary of major accomplishments of our project and an outlook towards future work.

CHAPTER 2. Simulation of Vehicle System Dynamics

Vehicle system simulation is one of the essential steps in the overall product development process, as it allows design engineers to fine tune the design parameters for its individual subsystems. System simulation also enables engineers to study the effect of selected parameters on the system through a display connected to the simulator, in order to adjust them to improve the performance. Dynamics of a subsystem are simulated using its mathematical representation, the general form of which is given by

$$\dot{y} = f(u, y) \tag{2.1}$$

where $\dot{y}$ is the slope of the subsystem and function f represents the dynamics of each subsystem using the state-space form. Function f depends on the system input u, and present state y of the system. The general state-space form of the models which have a linear effect on the dynamic behavior of the vehicle is given by

$$f(u_i, y_i) = A * y_i + B * u_i \tag{2.2}$$

where y_i is an N x 1 size vector representing the N states of the system at the present time, A is N x N state transition matrix that defines the coupling between various states of the system, u_i is the system input vector of size M x 1 and B is an N x M input matrix that defines the states to which the system input will be applied. For non-linear models, f does not have any generic representation and varies with different subsystems under consideration. However, for ease of application, non-linear subsystems may be modified and represented using the general form given in Equation 2.2.

RTS is performed by integrating f using numerical integration methods, where the output of the integration defines the state of the system after each time step. The same is considered useful for real-time analysis if the computation time for integrating one time step is less than the real-time constraint. In other words, given the state of the system y_i at time t_i, the time required to compute new state y_{i+1} at time t_{i+1} should be no longer than the difference between t_{i+1} and t_i, which is the actual time step being simulated. For an initial experiment, the methods were implemented in MATLAB, where it was observed that the computation time was negatively affected by the increase in the number of computations, which varies with the choice of numerical integration method, the size of the time step, and the number of physical states being modeled. It is important to note that though an equivalent C code may be faster than the MATLAB implementation, we would still observe similar trends, although a more complex method and model, and a smaller time-step would be needed. We describe below in brief the effect of each of these factors on the time taken per iteration in a linear model:

- Figure 2.1 and 2.2 shows the steps involved in computing new state of the system using RK1, RK2, and RK4, and the Adams-Bashforth, Adams-Moulton integration methods, respectively. The comparison of RK1, RK2, and RK4, shows that as we increase the order of the Runge-Kutta method there is an increase in the number of function evaluations, making the method computation-intensive. Also, we observe that the memory requirements to store previous iterations data remain approximately same for the three algorithms since they require data of only the previous time step to compute y_{i+1}. On the other hand, Adams-Bashforth and Adams-Moulton require fewer function evaluations at each time step, they need function evaluations at the previous two or three time steps to compute y_{i+1}. The 4^{th} order Adams-Bashforth needs function evaluations at time t_i, t_{i-1}, t_{i-2} and t_{i-3}. The 4^{th} order Adams-Moulton is a predictor-corrector method. It uses Adams-Bashforth to predict y_{i+1} and then uses function evaluations at time t_{i+1}, t_i, t_{i-1} and t_{i-2} to compute y_{i+1}. Thus for higher order systems these type of integration algorithms become resource hungry owing to the requirement of saving huge data from previous time steps. This in turn negatively affects the computation time.

- Sharp discontinuities in the model emphasizes the need for a smaller time step to accurately capture the behavior of the system. A smaller time step implies that the simulation

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

Figure 2.1 Numerical integration using Runge-Kutta methods

needs to do the same amount of work in lesser time as it did for a larger time-step in order to meet the real-time constraints. While the total work per iteration remains unchanged, each of the iterations has a smaller time budget.

- If the time step is kept constant, an increase in the number of states or order of the system increases the number of computations within each time step. In Equation 2.2, consider matrix-vector growth of $A*y_i$ for both a 4^{th} and an 8^{th} order system. For a 4^{th} order system the computation requires 16 multiplications and 12 additions and for an 8^{th} order system it requires 64 multiplications and 56 additions. As we increase the number of states in the system (N) the number of multiplication and additions increase on the order of $O(N^2)$ and $O(N^2\text{-}N)$ respectively. The number of computations involved in the multiplication of $B*u_i$ also follows a similar trend. While the total time allotted per iteration remains the same, the amount of computations performed per iteration increases as on the order of $O(4 \cdot N^2\text{-}2 \cdot N)$ with an increase in N.

Figure 2.3 compares the CPU computation time with varying time step and number of states using different numerical integration methods. The computation time does not include the time required to receive the steering wheel angle from the HIL, time to compute the position coordinates, and time to send these coordinates to the display monitor. Figure 2.3(a) shows the effect of reducing the time step on a 16^{th} order linear system solved using RK1, RK2,

Adams-Bashforth

$y_{i+1} = y_i + (55*f(u_i,y_i) - 59*f(u_{i-1},y_{i-1}) + 37*f(u_{i-2},y_{i-2}) – 9*f(u_{i-3} , y_{i-3})) * h / 24$

Adams-Moulton

$y_{i+1} = y_i + (9*f(u_{i+1},y_{i+1}) + 19*f(u_i,y_i) - 5*f(u_{i-1},y_{i-1}) + f(u_{i-2},y_{i-2})) * h / 24$

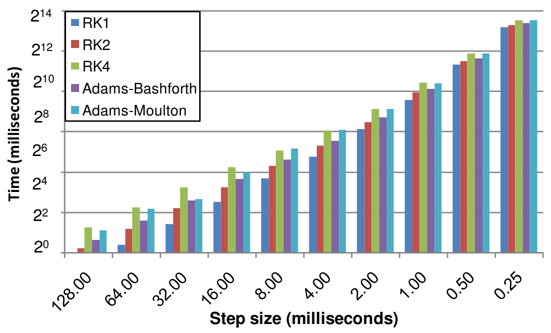
where

$f(u_i , y_i)$ = function value at time t_i	$f(u_{i+1} , y_{i+1})$ = function value at time t_{i+1}
$f(u_{i-1} , y_{i-1})$ = function value at time t_{i-1}	y_i = State of the system at t_i,
$f(u_{i-2} , y_{i-2})$ = function value at time t_{i-2}	y_{i+1} = RK1 approximation of $y(t_{i+1})$,
$f(u_{i-3} , y_{i-3})$ = function value at time t_{i-3}	h = Time Step

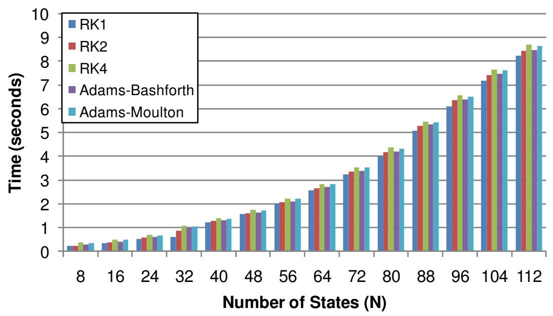
Figure 2.2 Numerical integration using Adams-Bashforth and Adams-Moulton Method

RK4, Adams-Bashforth, and Adams-Moulton algorithms. It was observed that as the time step is reduced, the time taken to simulate for five seconds increases, for a fixed set of design parameters. When the time step is reduced to .2ms the simulation fails to meet the real-time constraints for all the algorithms. Figure 2.3(b) shows the effect of increasing the number of states with a fixed time step of size 1ms. When the number of states are increased to 72, Adams-Bashforth and Adams-Moulton fail to meet the constraints as the overall computation time surpasses the real time of 5s. When the number of states are increased to 112, all the Runge-Kutta algorithms fail to meet the constraints. These results drive our research to alternate platforms for simulating more complex vehicle dynamics in real-time.

For our work, we considered an 8^{th} order steering valve and vehicle model. The vehicle model, as can be seen in Figure 2.3(a), is able to generate the simulation output in real-time in MATLAB. However, the steering valve model exceeds the real-time constraint by 3μs and forms the major computational bottleneck to run the real-time simulation. The parallelism involved in the matrix-vector multiplications of Equation (2.2) makes them a good candidate for acceleration using FPGA hardware. The computation of the position coordinates, on the other hand, is a two-step process with simple scalar multiplications and can remain in software. This forms a part of our system-level analysis. As mentioned previously, we developed a methodology to perform hardware design analysis. It involves analysis of the factors that affect the hardware design and involves the actual design generation followed by hardware simulation



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



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

Figure 2.3 Effect of step size and number of states on the CPU computation time for RK4 integrator

to check for functional correctness. For software implementation the aim is to minimize the time between the send and receive data commands to and from the hardware. Apart from the interface delay between the hardware and the host machine and the actual simulation run time which form a part of the hardware partition, the software design should be efficient with network communication and computation.

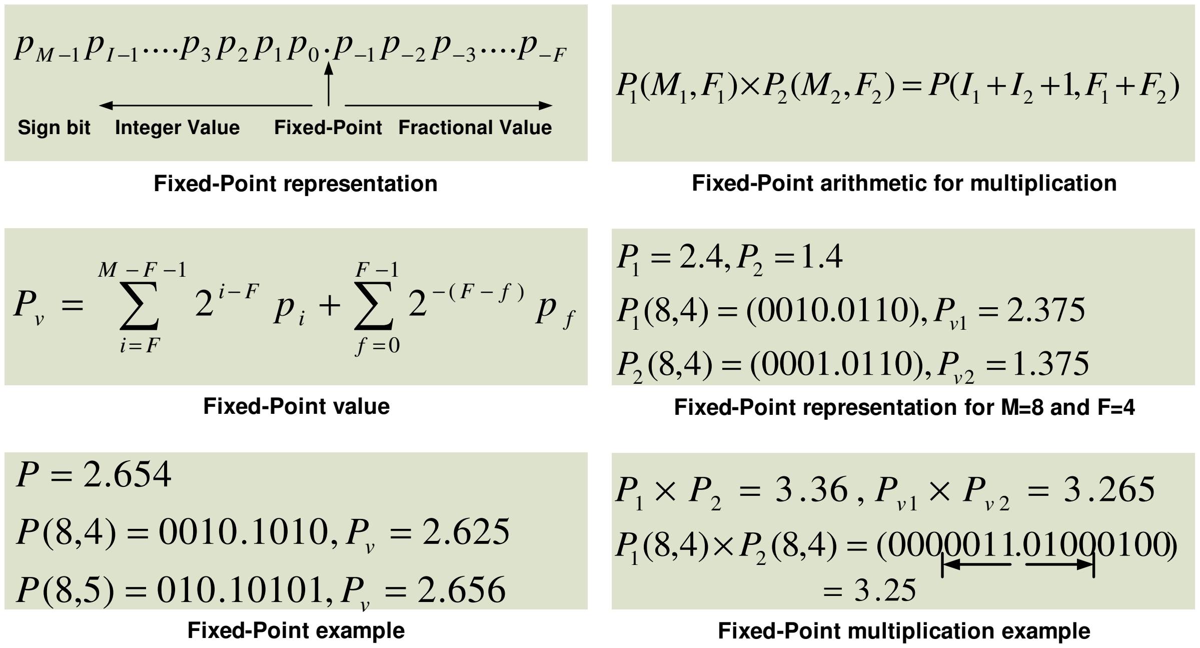
CHAPTER 3. Design Methodology

The first step of our methodology is the system-level analysis to determine the partitions and we consider factors that affect the partition of the design across hardware and software. The first factor is the computation time of different components of the simulation model and the second factor is 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 on the software. For example, a simulation model that involves a square-root operation followed by an expensive RK4 integration method, we obtain a partition which implements the square-root on the software and the integration method on the hardware. While considering the second factor on the same example, if the two partitions communicate only after a certain interval of time the partition might still be beneficial. However, if there is continuous exchange of data between the two operations then this partition will be quite expensive due to increased communication delay between hardware and software. Based on the components selected for either hardware or software implementation we first discuss the hardware partitioning followed by software partitioning.

3.1 Factors affecting hardware partitioning

After hardware/software partitioning, the implementation on the hardware itself needs to be partitioned across multiple FPGAs. The hardware/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 hardware design is thus generated based on variation in each factor with respect to another which we discuss in this section and then present the methodology taking into consideration different variations.

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 [52], described in Figure 3.1(a), which consists of fixed number of integer and fractional bits before and after the fixed-point. Given a signed number P(M,F), M is the total number of bits and F is the number of fractional bits, the integer bits I is equal to M-F-1 and p represents each bit of the number P in the binary system. The most significant bit (MSB) represents the sign bit. The integer bit position starts at 0 and progresses by 1 towards the left of the fixed-point and the fractional bit position starts at -1 and progresses by -1 towards the right of the radix. In binary number system, the weight of the each bit position is higher than the weight of the previous bit position by a factor of 2. So, weight of the integer bits grows such as 2^0, 2^1,...., 2^{M-F-1} whereas weight of the fractional bits grows such as 2^{-1}, 2^{-2},...., 2^{-F}. Thus, to obtain the value of a binary number given a fixed-point notation, each bit is multiplied by the weight associated with that bit position as shown by *Fixed-Point value* in Figure 3.1(a). The accuracy is thus determined by the number of I bits available whereas the precision is governed by the number of F bits selected for fixed-point representation. We explain this concept with an example in *Fixed-Point example* of Figure 3.1(a) for number P=2.654, given the total number of bits M=8 for different number of F bits. For F=4 bits and I=3 bits, leaving beside a sign bit, the fixed-point representation of the number is 2.625. As we increase the number of bits for F=5 , I=2 we achieve a closer fixed-point value of the number as 2.656. However, a further increase in the number of F bits will leave only a single bit for the integer value which will affect the accuracy. This also affects the accuracy/precision in the arithmetic operations involving the fixed-point operands. The operations may generate results of length greater than either of the operands. For example a fixed-point multiplication involving operands each of length M=8 bits, generate results of length 2*M-1=15 bits. To maintain uniformity in the way the operands and result of the arithmetic computations is represented on the hardware, we convert this result to a length of M bits. The required multiplication result lies in the first I


$p_{M-1}p_{I-1}\cdots p_3p_2p_1p_0.p_{-1}p_{-2}p_{-3}\cdots p_{-F}$
Sign bit
Integer Value
Fixed-Point
Fractional Value
Fixed-Point representation
$P_1(M_1,F_1)\times P_2(M_2,F_2)=P(I_1+I_2+1,F_1+F_2)$
Fixed-Point arithmetic for multiplication
$P_v=\sum_{i=F}^{M-F-1}2^{i-F}p_i+\sum_{f=0}^{F-1}2^{-(F-f)}p_f$
Fixed-Point value
$P_1=2.4, P_2=1.4$
$P_1(8,4)=(0010.0110), P_{v1}=2.375$
$P_2(8,4)=(0001.0110), P_{v2}=1.375$
Fixed-Point representation for M=8 and F=4
$P=2.654$
$P(8,4)=0010.1010, P_v=2.625$
$P(8,5)=010.10101, P_v=2.656$
Fixed-Point example
$P_1\times P_2=3.36, P_{v1}\times P_{v2}=3.265$
$P_1(8,4)\times P_2(8,4)=(0000011.01000100)$
$=3.25$
Fixed-Point multiplication example

(a) Fixed-Point notation (b) Fixed-Point arithmetic

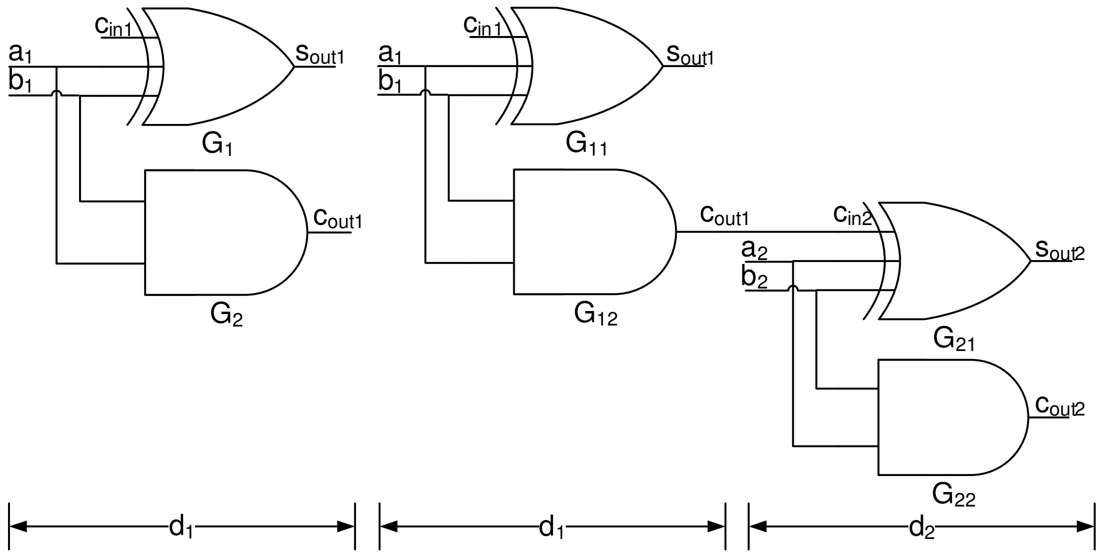
Figure 3.1 Fixed-Point representation

bits to left of the radix and first F bits to right of the radix. The truncation of 2F to F bits results in precision loss whereas reduction of 2I to I bits may result in a completely inaccurate result if I bits are not sufficient to represent the result. As seen in Figure 3.1(b), the M bits of multiplication represents 3.265. If we increase the number of F bits for each value, the result gets closer to 3.36. To accurately represent the result a minimum number of I bits are required.

For hardware implementation of the simulation model, the data values such as coefficient matrices, vectors, parameters and computation results are represented using this fixed-point notation. To determine the effect of the number of bits on the hardware RU we first look at the FPGA architecture. A conventional FPGA consists of an array of logic blocks, I/O pads and routing channels. The logic blocks are connected using routing channels which also terminate into I/O pads, connected to the actual pins on the FPGA device. Each logic block can accommodate only a specific number of input bits. As the number of bits is increased, the number of logic blocks being used increases and so does the routing between the logic blocks. The number of I/O blocks are limited by the number of pins and thus restrict the number of routing channels terminating on these pins. With limited number of these resources there is thus a restriction on the number of bits that can be accommodated on the hardware. A straightforward conclusion that can be drawn about the relation between accuracy/precision and space is that as we increase the number of bits to achieve between accuracy/precision in the simulation results, the space required increases.

The relation between time and space is based on the parallelism that can be explored 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 as minimum a time as possible. However, the parallelism comes at the expense of hardware resources. For example, if a hardware component "X" takes 10 cycles to compute the result and we need to implement 10 such components we have several options based on our requirement. First, we can concurrently run all the components thus obtaining the output from all in 10 cycles. This would be the fastest implementation but most resource hungry since each component would require independent resources. Second, we can serialize the computation such that when one component completes the execution only then the next one is executed. This would be the slowest and the hardware RU will be equivalent to that of single largest component. Third, we can pipeline the implementation such that a new computation is invoked every cycle. After an initial latency of 10 cycles, result will be obtained every cycle thus increasing the throughput. The hardware resource utilization will be somewhere between that of the earlier two implementations.

The relation between accuracy and time is based on variation in time with change in the number of bits. To explain this relation we take a simple example of addition. In digital logic, the hardware operations occur at the bit level using logic gates which are associated with certain amount of delay in transfer of data from input to the output port. Figure 3.2(a) illustrates the addition of two 1-bit numbers and Figure 3.2(b) illustrates the addition of two 2-bit numbers. In the former, we add two 1s represented by a single bit 1 and the delay associated in obtaining the final result is the delay through either of the gates, whichever is longer. In the latter, we add 3 and 1 represented by a 2-bit number 11 and 01 respectively. The delay d_1 is associated

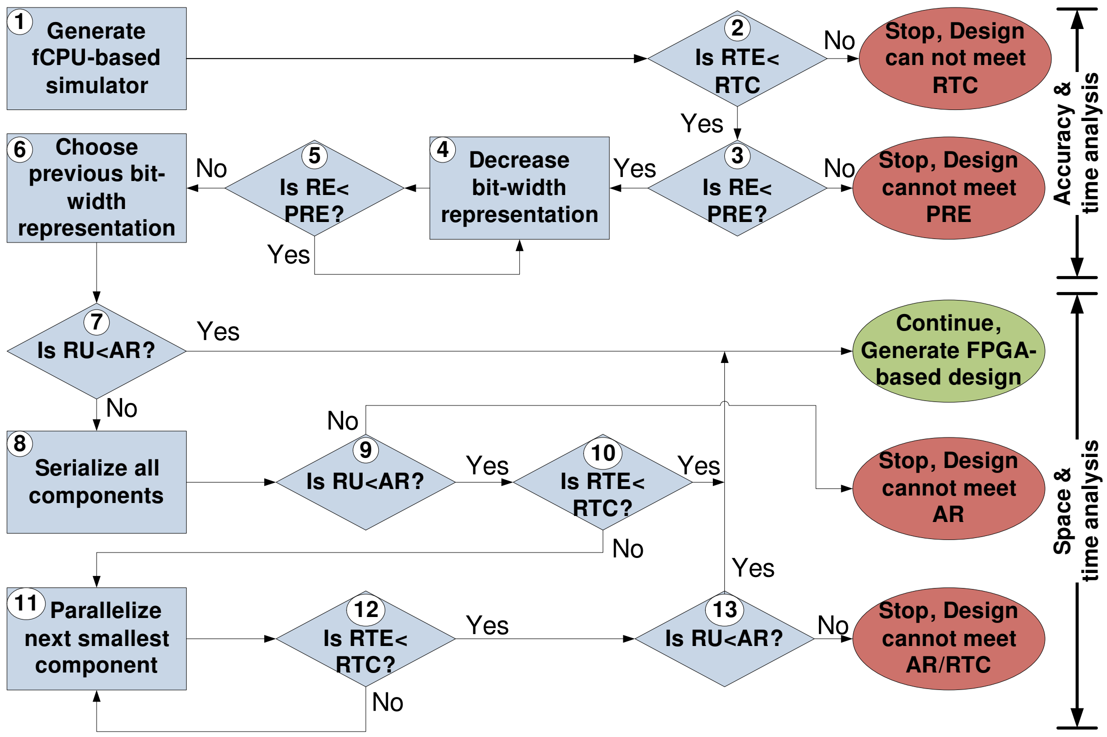


(a) Addition of two 1 bit numbers (b) Addition of two 2 bit numbers

Figure 3.2 Effect of number of bits on computation time

with the addition of first two least-significant bits. As per the addition rules, carry generated from the addition of these two bits has to be added with the bits in the next position. Thus, a valid result is available at the output of gates G_{21} and G_{22} only after delay $d_1 + d_2$ which increases with the increase in the number of bits. Thus, the number of bits affect the delay associated with generating the output of the individual components whose implementation vary with the number of bits.

The methodology to generate the FPGA-based simulator, based on the factors discussed, is shown in Figure 3.3. In the hardware design analysis phase, we first 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. The methodology has been proposed in the embedded systems literature [13, 27, 47] for Hardware/Software partitioning. It allows the designer to make and compare different design decisions for the physical system being simulated. It provides a systematic way of making changes in the hardware design and paves the way for automating the process for other vehicle systems. The steps shown in Figure 3.3 are performed manually and the automa-


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?
No
Yes
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
Yes
11 Parallelize next smallest component
12 Is RTE< RTC?
Yes
No
13 Is RU<AR?
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	

Figure 3.3 Heuristic approach for hardware partitioning

tion of this approach will form a part of our future research work. Except for the automation of *Simulation successful* step since that decision is based on the results from the Modelsim simulator (used for simulating VHDL/Verilog designs) which can be validated visually.

3.2 Hardware Design Analysis

The first step of the methodology involves selection of the numerical integration method and design of the CPU-based simulator. In this work we focus mainly on the simulation of vehicle systems using the RK4 integrator. Other integrators such as RK1, RK2, Adams-Bashforth and Adams-Moulton can also be used with our methodology; we leave this analysis to be part of our future research. The input to the hardware design analysis phase is thus 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) since the platform is pre-decided, with an assumption of aggressively parallelized design and maximum number of bits for fixed-point representation. Ideally, the value of PRE 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 Figure 3.3 we need a model which can give us an estimate of the required bit combination, time taken to complete a single iteration and the hardware RU. These estimates can be obtained by having a model which can emulate the FPGA computation process and we call this a fixed-point CPU-based (fCPU-based) simulator. We first design a software component library which contains equivalent software (MATLAB) representation of all the components in the hardware component library. Each functionality is implemented using the same techniques we used to implement them on the hardware. The functionalities 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 RK4 integration, we implement the algorithm for RK4 (as shown in Figure (2.1)) in MATLAB and use the same for the hardware implementation. Similarly, as will be explained in Section 4.2, we implemented square root on hardware using Goldschmidt's algorithm [42]. Instead of using the built-in MATLAB square root function we implement the Goldschmidt algorithm in MATLAB. In addition, the arithmetic operations in the modified implementation are also done using fixed-point notation and are parametrized for different bit combinations. Since the algorithms used are 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.

An alternate to generating an fCPU-based simulator is to directly generate the FPGA-based simulator, which uses the components from the hardware component library. These components are highly parameterized and read static data stored in the FPGA memory whose storage space varies with the change in the bit combination. The bit combination that satisfies the accuracy/precision and resource requirement for one component may not satisfy for a different component. Thus, after generating these components for a specific bit combination and a wrapper to connect them, it becomes necessary to validate the results from each component in the Modelsim. Without any criteria to select the bit combination, we may have to iterate through the entire process of selecting the bit combination, generating and validating data from each component in Modelsim several times before we achieve the correct bit combination. For complex systems with large number of components this process will involve a substantial amount of effort, thus the need for an fCPU-based simulator.

Step 2: A hardware equivalent simulation model is expected to speed-up the computation process via parallel architecture of the hardware. However, before we actually generate the design we estimate whether the hardware is capable of meeting the RTC even of the completely parallelized design. A parallelized design assumes that all the independent computations are implemented in parallel optimizing highly 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 estimation of time is a static process based on pen and paper analysis. The fCPU-based simulator gives us the hardware components required for FPGA-based simulator. For each of the independent components in the hardware component library, we also determine the number of cycles each component take to generate the output. We use the cycle information of each component to compute the number of cycles taken by the whole design to generate the output, taking into consideration the parallelism employed. Assuming different clock frequencies for the hardware, we can determine RTE of this design for these clock frequencies. If the RTE is more than the input RTC, the hardware will not be able to meet the RTC. This is because the RTE is being compared against the minimum possible time that an FPGA-based simulator will take by exploring all the parallelism in the model. If the time taken remains within the

RTC, we check for the RE constraint in 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 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 in the fixed-point format of length I+F bits. The converted values are equal or close to the true values if the bits are sufficient. The local truncation error due to each conversion and global propagation error due to previous conversions thus results in an error in the final simulation output after every iteration. Since the fCPU-based simulator is generated using algorithms used for FPGA-based simulator the conversion accurately models the hardware computation process and the error generated from this process can be considered as a close estimate of the RE that will be generated from the FPGA-based simulator.

To compute the RE between the fixed-point and the original output for fixed number of iterations we first compute the RE for each state in the system. The RE across all the states is averaged for every iteration which is further used to compute the RE across fixed number of iterations. For a fixed range of PRE and given the maximum number of bits for representation if the RE of the design fails to meet the PRE constraint we cannot proceed to the next step.

Step 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, at this point we would like to mention that the process of estimating the bit-combination using fCPU based simulator is highly dependent on the number of iterations we run the simulation for. 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 done assuming an aggressively parallelized model, which if implemented on hardware would utilize the maximum resources available. We optimized the design for time and determined if the model meets the RTC. We now optimize the design for space and determine if the model meets the AR constraint.

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 in turn increases the time taken to run a single iteration. Following our example in Section 3.1 where we considered 10 instances of a component "X". Now instead of all the computations being done in parallel we execute one computation followed by another. The serialization causes the output from the last computation to be generated in 100 (10x10) cycles though it will utilize the minimum resources since we use the same componenet but with different input. We can also pipeline the input such that it receives a new input every cycle. Since response to each input takes 10 cycles, the time taken to obtain the output for the last computation will be 19 cycles after the first input was received, as opposed to 10 cycles when all were implemented in parallel. The hardware RU reduces drastically though still more than a completely serialized design. Based on these optimizations, we obtain an estimate of the required resources and make a decision, whether or not, the design will fit on the hardware.

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 combination as shown in Figure 3.4. 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 Virtex-5 FPGAs of Xilinx can be obtained using the relation given in [33].

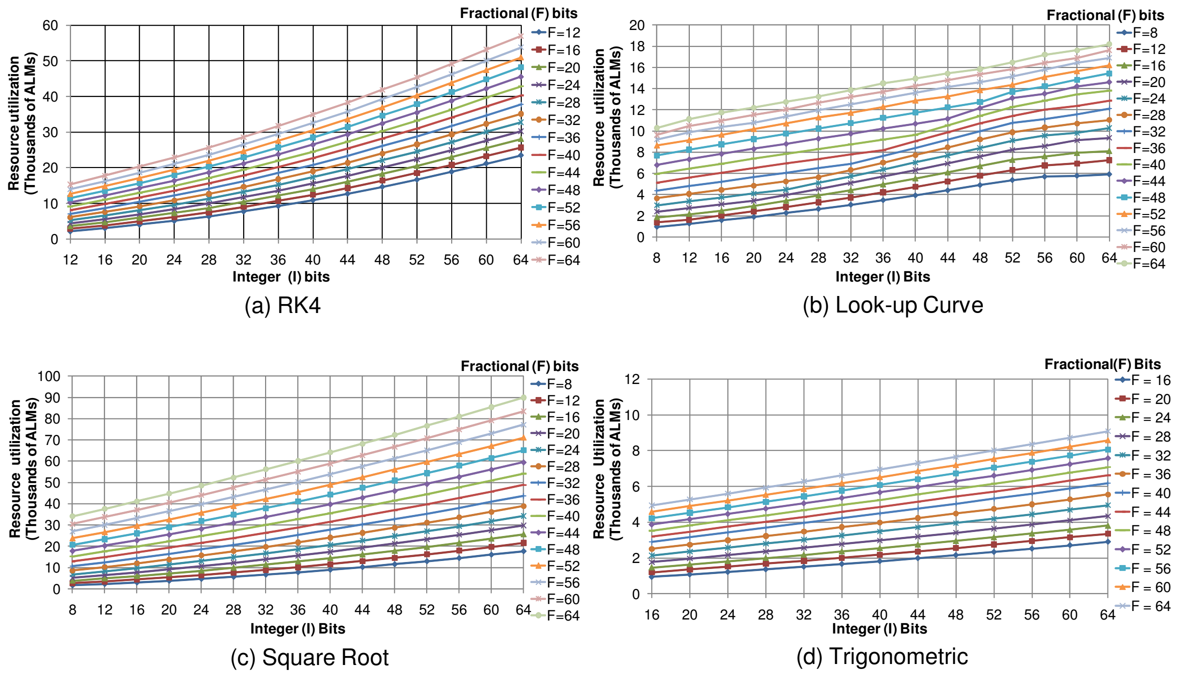


Figure 3.4 Hardware resource utilization for RK4, Look-up Curve, Square Root and Trigonometric Components

Step 7: For space analysis, we first check if with the selected bit-width combination from the previous step, the design meets the AR constraint. We use the fCPU-based simulator to determine the components that make up the FPGA-based simulator and use the graphs in Figure 3.4 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 combination. If it does not, we perform the space optimization and start with a completely serialized design in Step 8.

Step 8, 9 and 10: In Step 1 we started with an aggressively parallelized design which was 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 new computation starts either after completion of the previous computation or a cycle delayed. This process reduces the RU since the number of computations being done in parallel have been reduced. However, to obtain a lower limit on the hardware RU of the design, we start with a design that is completely serialized for which we again check whether with the selected combination, the design that has been optimized completely for space is able to meet the AR constraint in Step 9. Since we started with a serialized design, the estimate of RU is the minimum resources that a design is expected to consume and if the constraint fails, it is not possible to proceed further. If the constraint is met we then check if the new design meets the RTC in Step 10. As mentioned earlier, the serialization affects the RTE and if the completely serialized design meets 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 met, we still have the option to parallelize components in Step 11.

Step 11, 12 and 13: In Step 8, we made an assumption of completely serialized design and obtained the minimum resources that a design would consume. Since the RTC is not met, while still optimizing for space, we parallelize the component which consumes the minimum resources and thus results in minimum increase in the overall RU. We check for RTC in Step 12 and iterate through Step 11 and Step 12 until we meet the RTC. As discussed earlier as we parallelize components RTE reduces but the RU increases. So having met the RTC, we check for the RU constraint. If the constraint still fails, we cannot proceed further to generate the FPGA-based simulator else we use the automated scripts to generate the VHDL-design for the selected components with the selected combination.

To meet the real-time constraints the simulation results should be available at the host, for further processing, within or even less than the RTC. This is necessary because the time taken to complete an iteration includes the computation as well as communication delay. So far, we have been discussing the RTE as the time taken for computation on the hardware. However, we also need to include the time to transfer the data back and forth between the hardware and the host. The time only varies with the amount of data being transferred. We thus need an interface, which provides sufficient bandwidth and minimizes the latency in sending the data back and forth between the host and the hardware. Since the Hardware/Software partitions have already been decided, we know the amount of information that needs to be exchanged between the two partitions. We use this information to determine the required bandwidth of the interface and compare it with the bandwidth of the selected platform. We also compare the bandwidth of different interfaces available with different platforms and show that the choice of platform is the best option for our work. The interface details are also used in the software design phase explained in section 3.4.

3.3 Hardware Design Generation

An important aspect of the hardware design methodology is automatic generation of the design models based on different design decisions. The design decisions in this case include the selection of appropriate bit combination that meets the accuracy, time and space criteria. The advantage of having this automation is that the designer does not need to create a hardware design every time a design decision is changed. This allows the designer to focus on making the best design decisions without having to devote much time in creating the designs every time a change is done.

3.3.1 Design Generation

The VHDL design for each component is highly parametrized 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 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 create the design for all the components every time the bit combination changes. Thus, to automate the process of design generation, we designed the MATLAB scripts which take the bit combination and order of the system (if required) as input to generate the parameters file and VHDL design for components.

This step takes the bit combination, type of components involved and information about the serialization or pipelining of the components as input. Based on the bit combination and the type of components, we use these scripts to generate the VHDL design of different components. The wrapper, that intelligently connects these components is manually designed based on the information about serialization or pipelining of the components. Moreover, the actual decision whether the bit combination is sufficient, is made only after the next step i.e. design verification, where we visually compare the output from each component with the output from the CPU-based simulation model. If the verification fails we change the bit combination and start from step 7 of the hardware design analysis. The automatic generation of the components based on different bit combination thus reduces the amount of time the designer needs to spend to generate a new design and focus on analysing the effect of different design decisions.

3.3.2 Design Verification

An important check of functional correctness of the FPGA-based simulator is through Modelsim simulation. This is a design verification step, where 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 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 CPU-based simulator we generate the programming file and integrate it with the software design to run RTS.

Another reason that the design verification is important is that the process of creating a programming file for FPGA is a time consuming process. It includes synthesis, translate, map and place and route and the time taken may increase with the size of the design. It is highly unlikely that the hardware will work as expected in the first attempt and thus investing time to generate a programming file without verification is not recommended.

3.4 Software Design Analysis

The software design analysis phase is based on the platform selected in the platform analysis phase. This is 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

As we will see later, the system input i.e. the steering wheel angle, is assumed to change every 20 ms. The software should thus be able to perform the above tasks which includes the network delay in receiving the system from HIL, delay in sending the system input to the hardware and receiving the output from the hardware, computation time on the hardware and software, network delay in sending output to the VR display, within this real-time. The fast computations on the hardware cause the simulation to run faster than the real-time. 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 20 ms sends the output back to the software and stalls until it receives the new system input. On the software side, timer stops after sending the simulation output to the VR display so that the time to send the data over the network is included. At this point if the difference between stop and start timer is less than 20 ms, we invoke a sleep command to stall the software for the remaining amount of time i.e. 20 - (stop-start).

As the complexity of the physical system being simulated increases, the amount of work load for either hardware or software partition also increases. Considering the RTC, it thus becomes essential to develop an efficient software design that minimizes the time spent between the start and stop timers, apart from the hardware computation involved.

CHAPTER 4. Hardware Component Library

The IP core components provided by the FPGA vendors for non-linear functions, offer limited support with respect to type and length of input data. For example the top two FPGA vendors, Xilinx and Altera, both provide an IP core for square root operation. However, while the former supports only 48-bit wide input, the latter supports 256-bit wide integer input. Similarly, for trigonometric functions, Xilinx supports only 48-bit wide input whereas Altera doesn't provide such support. In addition, usage of IP cores takes away the benefit of design portability. A design developed using Altera's IP cores is not easily translated to a Xilinx platform and vice-versa. To overcome these limitations, we developed our own components for non-linear functions which support 128-bit (maximum of 64 bits for I and F each) input widths that are portable across different platforms.

Our current hardware component library provides support for look-up curve, square root, trigonometric functions, and a RK4 integrator. In this chapter, we discuss the algorithm and architecture of these components, compare our approach with previous implementations in the research literature and list the advantages offered by our implementation.

4.1 Look-up Curve Component

4.1.1 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 linear interpolation method given by the equation below to obtain this estimate.

$$Estimate = \left(\frac{Y_2 - Y_1}{X_2 - X_1}\right) * (input - X_1) + Y_1 \tag{4.1}$$

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

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

$$Slope = \left(\frac{Y_2 - Y_1}{X_2 - X_1}\right) \tag{4.2}$$

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

$$index = \left\lfloor \frac{input - X_0}{\Delta X} \right\rfloor \tag{4.3}$$

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 $\Delta\ X$ 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 *address* at which to access the two tables.

4.1.2 Implementation

In a five stage pipelined architecture shown in Figure 4.1, a valid computation starts when it receives a *start* signal. 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 Equation (4.3). 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 formula given below and also compute the difference *input* - X_1.

$$X_1 = X_0 + \Delta X * index \tag{4.4}$$

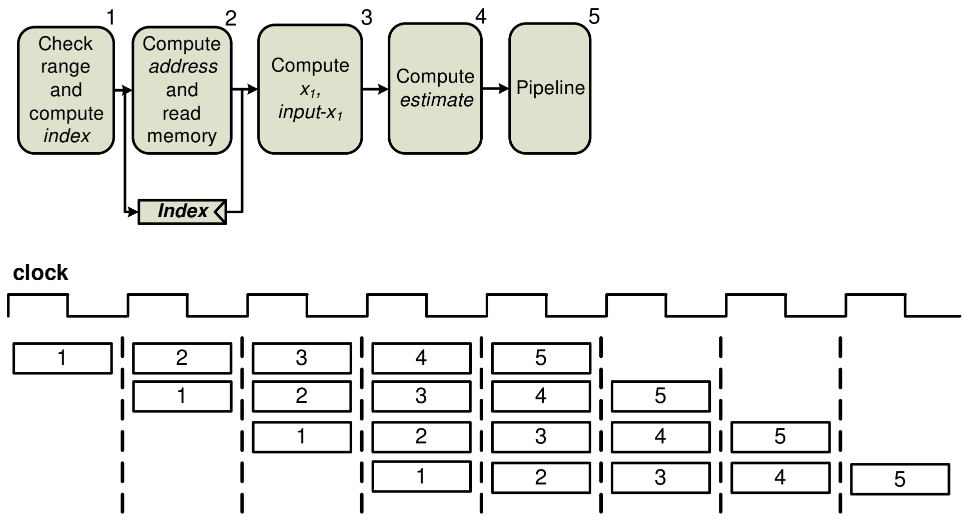

1
2
3
4
5
Check range and compute *index*
Compute *address* and read memory
Compute x_1, *input-x_1*
Compute *estimate*
Pipeline
Index
clock

Figure 4.1 Pipeline of Look-up Curve Component

In the fourth stage, we have all the values to execute Equation (4.1). The value of *input* - X_1 is available from the previous stage, *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 fifth stage. The pipelined implementation can accept a new *input* value every cycle. After an initial latency of 5 cycles, our component generates new output every cycle.

An important optimization achieved by splitting the look-up tables into two is that we save a delay of two cycles in computation of the *estimate* which will be explained in Section 4.3. Further, the implementation is highly parametrized 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 of ΔX in Equation (4.3) is implemented by saving its inverse in this file. When the bit combination changes, the auto-generation scripts for VHDL design generate these constants and look-up tables in the fixed-point format for the selected combination.

4.2 Square-Root Component

The algorithms for hardware implementation of square-root fall in two categories - subtractive and multiplicative [28], [41], [4]. Subtractive or direct methods are based on the conventional procedure of computing square-root by hand, where each bit of the result is computed in one clock cycle. This method is efficient for small number of input bits but the initial latency is very high for higher number of input bits, which is 128 in our case. 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 choose the latter category for our implementation. The Newton-Raphson method involves dependencies between its successive operations causing an uneven pipeline structure. We thus use the Goldschmidt algorithm [42].

4.2.1 Principle

The square root function is implemented using Goldschmidt algorithm [42] which is efficient in computing 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 x 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, square root of which is obtained from a look-up table that holds pre-computed square root values. The square root of a is determined using Goldschmidt algorithm. The square root of the original number is thus the product of the two square root values. The Goldschmidt algorithm is a three step process, described in Equation (4.5) where x_0 and y_0 are set to the initial guess value a. When the process is executed for few iterations as x_i converges to 1, y_i converges to $\sqrt{a}$.

$$r_i = (3 - x_i)/2 \tag{4.5}$$

$$x_{i+1} = x_i * r_i * r_i \tag{4.6}$$

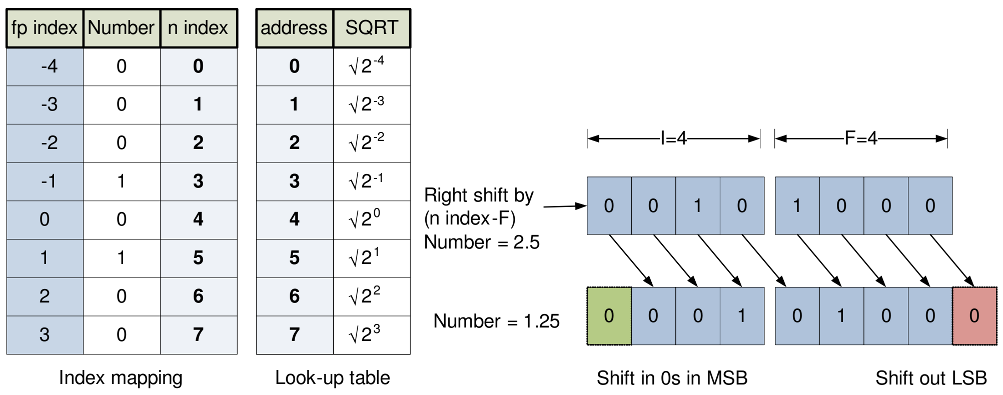
$$y_{i+1} = y_i * r_i \tag{4.7}$$

4.2.2 Implementation

The look-up table implementation is based on the bit combination selected for FPGA implementation. Given the number of I and F bits, the maximum and the minimum number that can be represented in the 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}, ..., 2^{0}, ..., 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 former is the index interpreted by the hardware for any binary number and also the address at which to read the look-up table. The latter 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.

Figure 4.2 explains the methodology to obtain the number 2^n, its square root and *a* for computing square root of *number* 2.5 with a bit combination of 4 and 4. In Figure 4.2(a), the table on left hand side gives binary representation of the number along with *n index* and *fp index* values. The table on 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 *n index* and *fp index* corresponding to the first occurrence of 1 from the most significant bit (MSB). The 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*. Figure 4.2(b) generates a number close to 1 by shifting the *number* such that the first occurrence of 1 from MSB now lies at the 0^{th} position of I bits. The number of bits to shift is thus computed by subtracting F bits from the *n index* value. If the input is less than one, *n index* will be less than F bits and the negative difference value will shift the input left else to the right. In this case *number* is shifted right by 5-4=1 bit.

During first stage of the pipelined architecture, we compute the address to access the memory, which stores the look-up table. The address is pipelined to be used in the later stages to read the memory. Initial guess value *a* for the Goldschmidt algorithm is also computed in the same stage. r_i is computed in the second stage followed by computation of x_{i+1}, y_{i+1} in the third stage. The Goldschmidt algorithm is executed for five iterations resulting in total run time of ten cycles. Since the output from Goldschmidt is not available until end of the eleventh stage, a valid read enable signal is sent to the memory in the tenth stage with the pipelined address value computed in the first stage. By the end of eleventh stage we receive



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**

Index mapping

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	$\sqrt{2^{3}}$

Look-up table

(a) Determine 2^n and it's square root (b) Determine a

Figure 4.2 Methodology to obtain 2^n, its square root and a

the square root of 2^n from the memory and the square root of a from the algorithm and obtain the product of the two in the final stage. The output is thus available at the end of twelfth stage. With a pipelined architecture after an initial latency of twelve cycles a valid square root value can thus be obtained every other cycle.

4.3 Trigonometric Function Component

4.3.1 Principle

The trigonometric function is implemented using linear-interpolation method described in Equation (4.1). However, the implementation is slightly different from the one used for look-up curve component. With 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, increased number of data points cost more in terms of resource usage.

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 trigonometric function we explore the symmetry among the function values and generate a look-up table that contains the sin values for 1250 equally spaced points between the interval 0 to $\frac{\pi}{2}$. The cos value is generated from the same table using identity cos(x) = sin($\frac{\pi}{2}$ -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 different sign magnitude. To determine the sign magnitude, we first reduce the input to the component in the range 0 to $2\times\pi$. We compute floor of the value obtained by division of the number with $2\times\pi$. The integer result obtained is multiplied again with $2\times\pi$ 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 magnitude depending on whether it is computing sin or cos and reduces the input to the range 0 to $\frac{\pi}{2}$. To compute the inverse trigonometric function, we generate a look-up table that contains the inverse sin values for 1250 equally spaced points between the interval 0 to 1. The cos value is generated from the same table using identity acos(x) = $\frac{\pi}{2}$ - asin(x).

For a look-up table approach with 1250 points, the principle described in Section 4.1, would need two such tables. If the number of bits selected is not huge, it may still be possible to implement a two table approach; however with increase in the number of bits, the resource usage will be tremendous. We thus use a single table approach which adds 2 cycle delay to the computation time obtained from the principle in section 4.1.

We make two assumptions for the input that is sent to this component for computing the trigonometric function. First, it is always a positive value. Second, it is in the range of 0 to $2\times\pi$. A number greater than $2\times\pi$ is converted to a number within this range. For a negative input value, we compute 2's complement of the number and send the modified value to the component. For sin function, we restore the sign by taking 2's complement of the output since sin(-x)=-sin(x). For a cos function, cos(-x)=cos(x), second 2'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 further reduction of input to the range 0 to $\frac{\pi}{2}$ is not required. So in the first stage we register the input and compute the index. For computing trigonometric functions, in the first stage a further reduction to the required range is done before computing the index. The index from the first stage is used to compute the address of two consecutive data points in the second stage. The third stage does the same arithmetic, However in the fourth stage, instead of executing equation (4.1), we compute the *difference* of the two function values read from the memory and also pipeline the lower index data point until sixth stage. The division of *difference* by constant ΔX is implemented by multiplying with its inverse to obtain the slope estimate in fifth stage. In the sixth stage, function value is obtained by computing the product of the result from the fifth stage, slope estimate with the result from the third stage, *input* - X_1 and added 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 seventh stage.

CHAPTER 5. Application of the Methodology on an 8^{th} Order Vehicle System

We apply the methodology discussed in Chapter 3 to generate a FPGA-based simulator for an 8^{th} order vehicle system. The steering valve dynamics that simulates the delay between rotation of the steering wheel and actual movement of the tire is very sensitive to change in the steering wheel input. The dynamics of this system have a very small time constant, and with RK4, the system is numerically unstable for integration step much larger than time step h_{valve}, of 10 μs. The vehicle dynamics that simulates the overall movement of the vehicle, can be captured with a time step, $h_{vehicle}$, of the order of few milliseconds. With such a small value of time step for the steering valve model, when the vehicle system simulation is implemented on MATLAB, simulation output fails to meet the real-time constraints. The vehicle dynamics when simulated by itself meet the constraints. In this Chapter we first discuss the dynamics of the vehicle system and then apply the methodology discussed in Chapter 3.

5.1 8^{th} Order Vehicle System - Steering Valve and Vehicle model

The steering valve dynamics is described in detail in [18]. It 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 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 Equation (2)[43].

$$\dot{p} = \frac{\beta}{V} \times (\Delta Q_v) \tag{5.1}$$

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 angular position of steering wheel (A_s) and gerotor motor (A_m) given by

$$rdel = A_s - A_m \tag{5.2}$$

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

$$A_m = \frac{-q_1}{I_g} \tag{5.3}$$

where q_1 is derived state of the system and I_g is gerotor inertia. Since 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 relation given below:

$$\begin{aligned} sqrt &= \sqrt{\left(\frac{2 \times abs(pi - pf)}{\rho}\right)} \\ Q &= A(\Theta) \times Cd \times sqrt \times sign(pi - pf) \end{aligned} \tag{5.4}$$

where $A(\Theta)$ is the valve opening area, p_i and p_f are the inlet and outlet pressure at valves, Cd 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 is explained in [19].

Figure 5.1 shows the architecture of the vehicle system. The steering valve model consists of three units: valve opening area, orifice flow rate, state-space solver. The vehicle model consists of two units: trigonometric function and state-space solver. The non-linear dynamics of the steering valve model reads A_s from HIL and previous state of the valve to compute system input for the state-space solver. The solver implements a numerical integration method, with a time step h_{valve}, to compute new state of the valve. The state variables model different attributes of

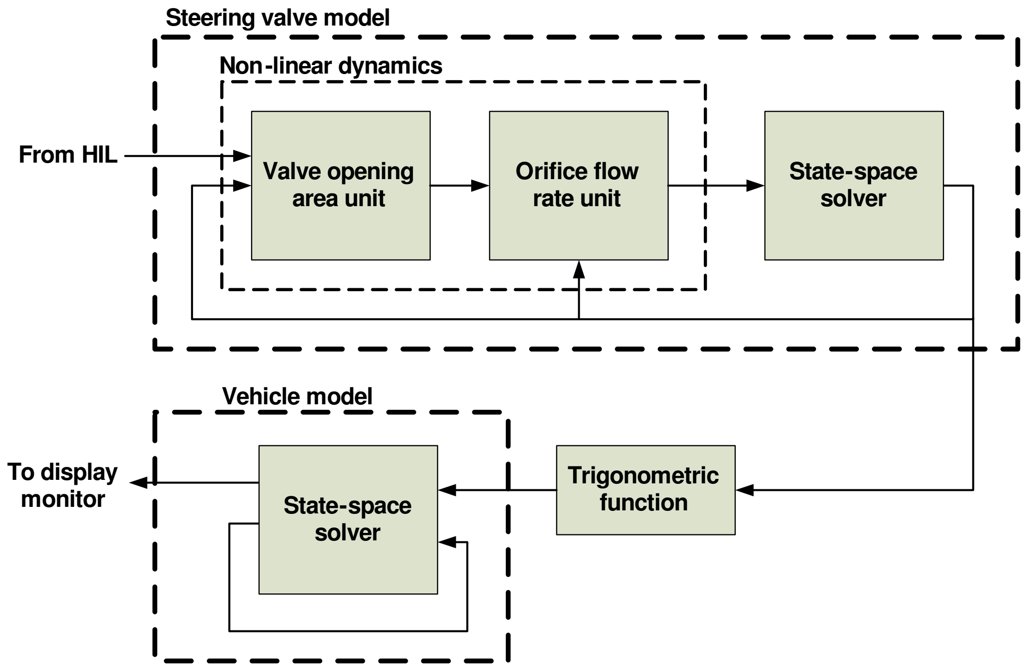


Figure 5.1 Architecture of the vehicle system

the valve and one of the states is used to compute system input for the vehicle model using a trigonometric function. The solver for vehicle model also implements the numerical integration method, with a time step h$_{vehicle}$. It reads the previous state of the vehicle and 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 thus 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.

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 valve opening area unit, along with the present state y_i of the system and computes the flow rate through each valve using equation (5.4). The inlet and outlet pressure values at the four desirable valves are determined from the 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.

The state space solver for both the models perform actual simulation process by numerically integrating the models at their respective time steps. Assuming that the system input for the valve model A_s, is received every S seconds. This implies that we want to determine the final state of the vehicle after S seconds. Though, in real-time scenario, the two systems run in parallel. But for simulation purposes, we shall run the steering valve first and then the vehicle model. The steering valve model is thus simulated for $\frac{S}{h_{valve}}$ iterations followed by the vehicle model which is simulated for $\frac{S}{h_{vehicle}}$ iterations. The output of either the last iteration of the steeing valve model or the average of all the iterations over S seconds is fed to the vehicle model whereas the output of the last iteration of the vehicle model represent the state of the vehicle after S seconds.

The models integrated by the state space solver, are in general form of state space representation of a linear system, given by equation (2.2). Though the state space representation of non-linear systems varies with systems under consideration, we modify the non-linear steering valve model to be represented in this form. Author [18] thus developed a set of equations using relation in equation (5.1) and generated the required coefficient matrices and vectors for state space representation of the form in equation (2.2) for the steering valve model.

The coefficient matrix (A) and state variable vector, y_i are shown in Figure.5.2(a). In A, the first four rows model the hydraulics dynamics where β_1 to β_4 are the fluid bulk modulus of the volumes V_1 to V_4. C_{L1}, C_{L3} and C_{L2} are the leakage flow coefficients for respective volumes, 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

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

(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$_i$

Figure 5.2 Coefficients and variables for Steering Valve and Vehicle model

by p_1 to p_4.

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

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

The coefficient matrix B and the system input vector are shown in Figure.5.2(b). In the input vector Q_{ol1}, Q_{ol2} and Q_{ol3} are flow rates through left end of the cylinder, Q_{or1}, Q_{or2} and Q_{or3}, are flow rates through right end of the cylinder. The rows corresponding to flow rates Q_{ol3} and Q_{or3} in the input matrix B are zeros which implies that these flow rates do not affect the final state of the steering valve system. This explains the reason for four instead of six pressure values in the state variable vector associated with the four valves.

5.2 Application of the Methodology

Based on the criteria described in Chapter 3 for system-level analysis, we implement the computation intensive part of the model i.e. numerical integration method in the hardware.

The parallelism in the computations involved in RK4 make it an ideal candidate for FPGA implementation whereas the method to compute the position co-ordinates involves relatively simpler execution and can be implemented on the software. However, the frequency of communication between the two models allows the following two Hardware/Software partitioning schemes:

- The implementation of both the computation intensive models, which can be efficiently parallelized and pipelined in hardware, while keeping the computation of the position coordinates in the software.
- The implementation of just the steering valve model in hardware, since the vehicle model can easily meet the real-time requirements with a software implementation. The computation of the position coordinates remains in the software as well.

We discuss in detail the first approach and apply the methodology described in Figure 3.3 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 5.4.

Step 1: The input to the Step 1 of our methodology is the CPU-based simulator, .01% of PRE, RTC of 10 μs for the steering valve model, 2ms for the vehicle model and 101,760 x 2 ALMs of AR (Dual FPGA). To generate an fCPU-based simulator, we use a CPU-based simulator that implements the dynamics of the vehicle system and find the functionalities that have an equivalent component in the hardware component library. In our vehicle system the valve opening area unit uses a linear-interpolation method which can be implemented using look-up curve component on the hardware. The orifice flow rate unit involves multiplication and square root functions. The multiplications are implemented using hardware multipliers available on board and the square root function is implemented using a component from the library. The state space solver for both the models uses RK4 component explained in next section. After having decided the components, all the functionalities in the CPU-based simulator are replaced with the selected components from the software component library to obtain a fCPU-based design.

Step 2: For this work we do not consider the variation in time with respect to the number of bits. We assume that the time taken by the components remain constant as the bit combination changes. Table 5.1 shows the time taken by each component that constitute the FPGA-based simulator. To estimate the time taken for a completely parallelized design, we first list down the components in the order required to complete a single iteration of the steering valve and vehicle model. We need four look-up curve components followed by a pipelined square root component for 6 orifices. The output of the square root component will feed the RK4 component for the steering valve which would then drive the RK4 component for the vehicle. The output from all the look-up curve components will be available in 5 cycles. To implement Equation 5.4 for 6 orifices, it will take a cycle to compute the 2s complement of the negative number, 12 cycles to compute the square root, a cycle to further complement the result of the square root and a cycle to implement the two multiplications. Thus, the output from all the orifices will be available in 21 cycles. An RK4 component for steering valve model takes 169 cycles and that for vehicle model takes 113 cycles. An additional 7 cycles are consumed by a trigonometric function at the input of RK4 component of the vehicle model to obtain the angular piston displacement. Thus, a single iteration of steering valve and vehicle model takes 315 cycles. For a design running at 100MHz the time taken will be 3.15μs, for a desing running at 75MHz the time taken will be 4.2μs and for a design running at 60MHz the time taken will 5.25μs which when compared with 10μs meet the RTC.

Step 3: Ideally the PRE is determined when the model is built based on the amount of error it is capable to tolerate. For our work the PRE is set to .01%. Given the maximum bit-width combination of F=64 and I =64, the plots for relative error in Figure 5.3 show that the PRE constraint is easily met with this combination.

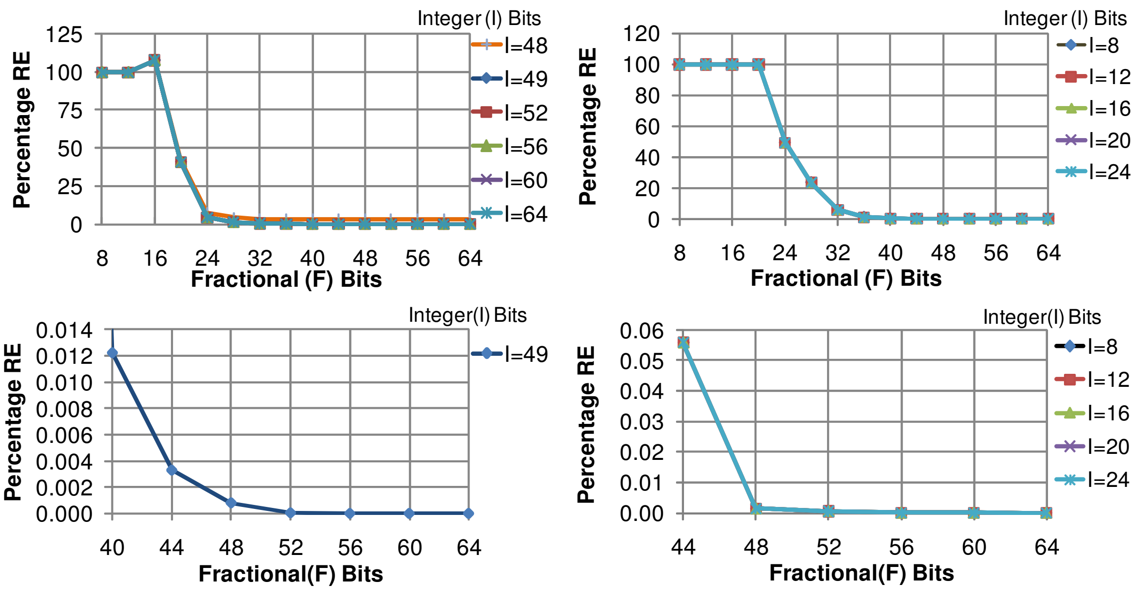
Step 4, 5 and 6: We further reduce the bit-width combination by estimating the relative error for different combinations. We start with the maximum bit-width of 128 for which the relative error is close to zero. We first estimate the number of I bits for which the relative error remains within the given PRE value and then reduce the number of F bits until it meets the

Table 5.1 Cycle Count by Individual Hardware Components

Component	Cycles Count
RK4 Vehicle	113
RK4 Valve	169
Look-up Curve	5
Square Root	12
Trigonometric	7

criteria. The Figure 5.3(a) shows the relative error in the output of non-linear steering valve model for I ranging from 64 down to 48 with different values of F. As we reduce the number of F bits the relative error increases until F=16. However, for values of F smaller than 16, the relative error is close to 100%. This is because the data values are so small that they are converted to zeros due to insufficient bit-width. As a result the relative error shows a drop to 100% for values of F less than 16. The number of F bits for which the relative error reaches an error value close to the given range is thus estimated to be M=91 and F=42. Similarly for the linear vehicle model Figure 5.3(b) shows the relative error in the output for I ranging from 24 down to 8 with different values of F. As we reduce the number of I bits, the overlapping plots imply that the relative error remains constant whereas it increases as we reduce the number of F bits. The number of F bits for which the relative error reaches an error value close to the given range is thus estimated to be M=54 and F=46.

Step 7: Assuming the completely parallelized design, to obtain the estimate of hardware RU for the selected combination from the previous step we refer to the graphs in Figure 3.4 and Figure 5.4. The RU is computed in terms of Altera's ALMs. The steering valve model needs four look-up curve, six square root, one RK4 component and a trigonometric function at its output, the total RU is approximately 98K ALMs. The vehicle model needs one RK4 component, the resource usage for which is approximately 27K ALMs. The total resource usage is thus estimated upto 125K ALMs which is within the range of the AR of 203.52K ALMs. The automated scripts take the order of the model and bit-width combination as input. The scripts generate the parameters file, look-up tables for each component with data in fixed-point



(a) Relative error for Steering Valve Model (b) Relative error for Vehicle Model

Figure 5.3 Relative error in the output of Steering Valve and Vehicle Model

format with bit-width of M. A wrapper is then designed to connect the look-up curve, square root, RK4 components.

To check the design for functional correctness we run the MATLAB version of design for few iterations and log the output after each computation in a file. We also convert the value to equivalent fixed-point representation using the selected bit combination, in hexadecimal format. The hardware simulation for Modelsim is also run for atleast the same number of iterations which shows the results in hexadecimal format. It thus becomes easier to debug and locate a faulty component resulting in a mismatch. With the initial bit-width combination the Modelsim simulation output does not match the MATLAB output because of insufficient number of F bits to represent the small values generated during the computation. We increase the number of F bits and go back to *Step 7* and apply our methodology thereafter. The final bit-width combination which generates the Modelsim simulation output close to the MATLAB output is M=96, F=47 for steering valve model and M=56, F46 for the vehicle model. The hardware RU for this combination is 113K ALMs for the steering valve model and 30K ALMs for the vehicle model which meets the RU constraint.

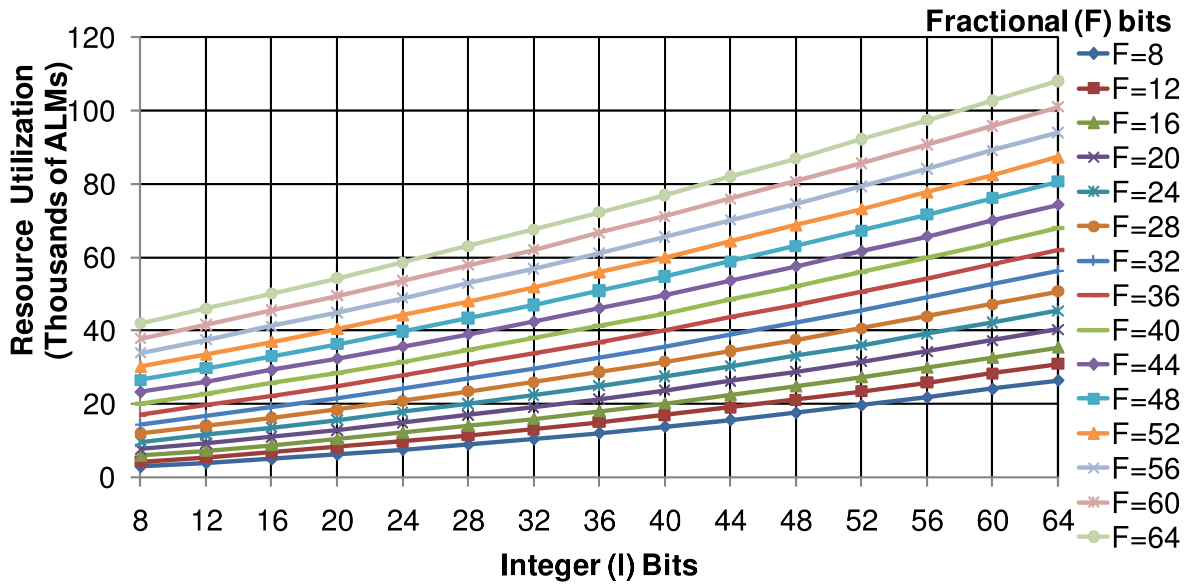


Figure 5.4 Hardware resource utilization for RK4 component for the vehicle model

To estimate the bandwidth requirement of the design we use Figure 5.5 which plots the bandwidth requirement of the vehicle model, with just the RK4 component, and it shows that the required bandwidth increases with the increase in the order of the model. We further compare it with the bandwidth offered by different interfaces available on Xilinx and Altera boards. Though PCIe comes across as a good option for the 8^{th} order model, the interface will not be beneficial for higher order models where bandwidth requirement is close to its offered bandwidth. The required bandwidth is based on the time taken to generate the output such that as the time taken increases the required bandwidth decreases. The complexity and order of the model increases the time taken to generate the output as a result there is a decrease in the required bandwidth. So, the vehicle system with just the vehicle model when combined with the steering valve model, the required bandwidth is expected to decrease owing to increase in the time required to generate the output due to added complexity.

To obtain the required bandwidth for the steering valve and vehicle model we assume the design to be running at 100MHz. The steering valve model needs 128 bits for a reset signal, 128*N bits to define initial state of the system and 128 bits for the system input. Though it

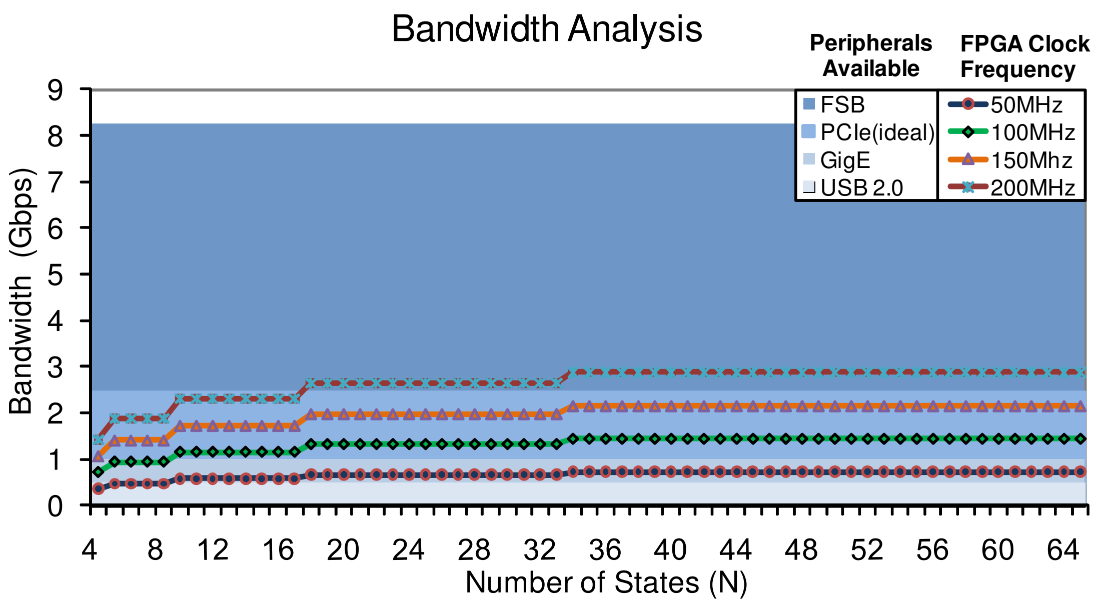


Figure 5.5 Bandwidth analysis

needs the first two values only once, when the simulation triggers, we compute the bandwidth considering the maximum requirement. We briefly discussed in section 3.4 that the hardware stalls after simulating for 20 ms until it receives a new system input. So, even though the hardware does the computations faster than 20 ms, it does nothing until 20 ms have passed. We thus use the time to estimate the bandwidth as 20ms. The required bandwidth for N=8, from CPU to FPGA is calculated using formula $\frac{1280*100MHz}{20ms}$ and is estimated to be 64Kb/s. The vehicle model sends eight states, 128*8 and a counter value, each 128 bit wide, also every 20ms. Thus, the required bandwidth from FPGA to CPU is estimated to be $\frac{1152*100MHz}{20ms} =$ 57.6Kb/s. For this work we use XtremeData platform which provides a bandwidth of the order of 8.28 Gb/s (1.035GB/s). Figure 5.5 gives a brief comparison of the bandwidths offered by different platforms and shows that with the given bandwidth requirement most of the interfaces such as PCIe (2.5 Gb/s), USB (.48 Gb/s), GigE (1 Gb/s), FSB (1.035 GB/s) shall be able to satisfy the demand. However, it is the resources which form a bottleneck for this design. Table 5.2 compares the RU with the AR on different Stratix-III and Virtex-5 FPGA boards. For

Table 5.2 Comparison of FPGA devices

Device type	ALM	LUTs	Equivalent ALMs[1]	ALMs required for Design
Stratix 3SL200	79.5K	x	x	143K
Stratix 3SE260	101.76K	x	x	143K
Stratix 3SL340	135.2K	x	x	143K
XC5VLX220	x	138.24K	115.2K/76.8K	143K
XC5VLX330	x	207.36K	172.8K/115.2K	143K

Xilinx devices, with LUTs as basic building blocks, equivalent number of ALMs are obtained by using relation defined in [33]. The table lists the largest devices in terms of basic building blocks for Stratix III and Virtex-5 family. Though all of them fail to meet the space constraint, the XtremeData platform with two Stratix 3SE260 boards becomes an appropriate choice. It will allow us to add more components to the design without reaching the limit soon.

5.3 State-space solver - RK4 Integrator

The actual simulation process is performed by solving state-space representation of the system using RK4 integration method [5] and the steps involved are described in Figure 2.1. The function f is the state space representation of the system described in Equation (2.2) and we use this representation to formulate the dynamics of the steering valve and vehicle model. The steps involved in the computation of y_{i+1} is sequential as the computation of new slope (k_2, k_3, k_4) is dependent on the previous slope (k_1, k_2, k_3) respectively. Since we cannot parallelize them we explore the parallelism within each computation. The RK4 integrator for FPGA-based simulator consists of three components: RK4 controller, system dynamics and system output, where the latter two, implement the actual dynamics of the integrator and the former implements a state machine to control the flow of information between the other two. As shown in data-flow diagram through a single iteration in Figure 5.6 each iteration is divided into four stages. During the four stages, 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

[1]For Xilinx, 1ALM=1.2LUTs/For Altera, 1 ALM=1.8LUTs

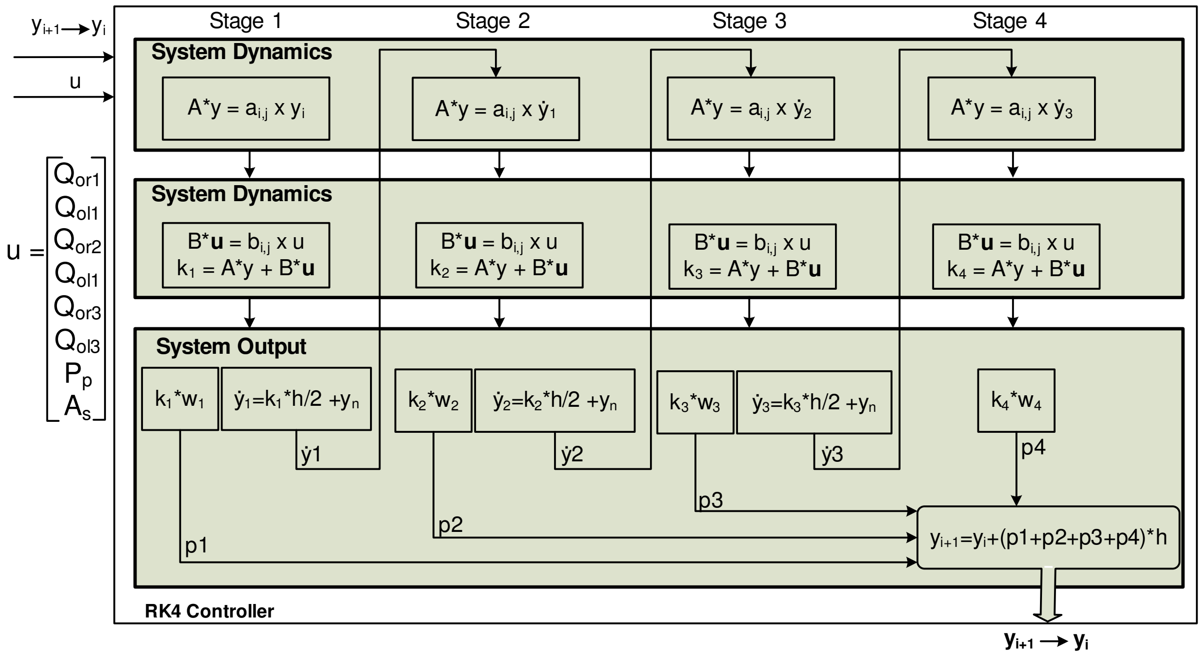
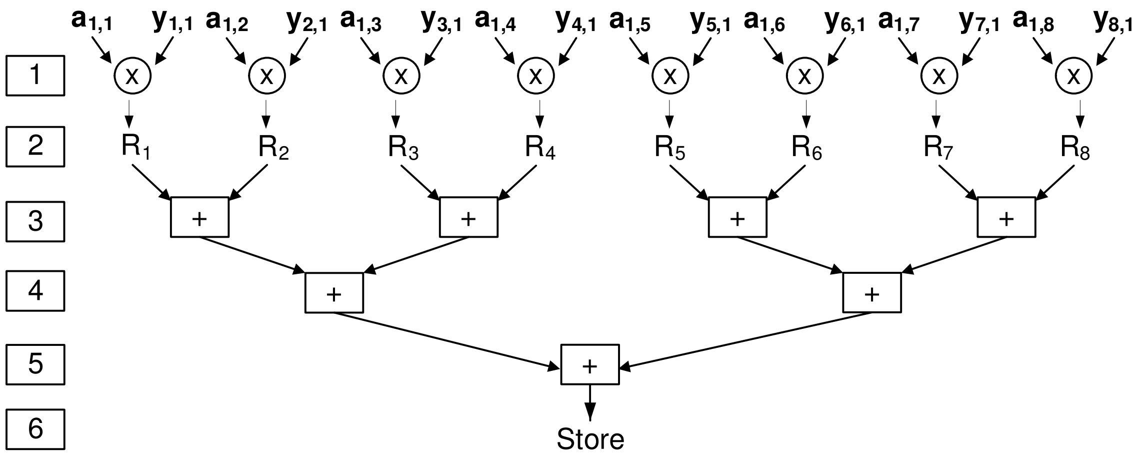


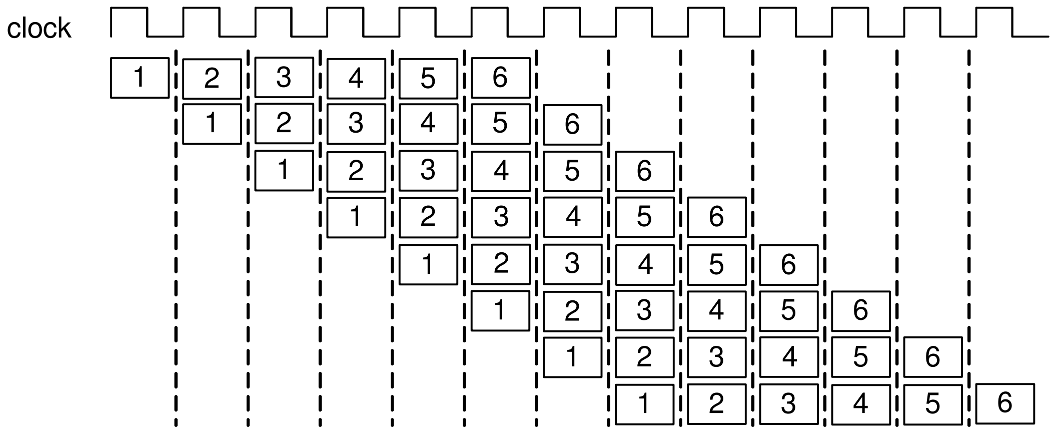
Figure 5.6 Data-flow diagram for RK4 iteration

computes 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 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 FPGAs parallel architecture 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 parallel 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 x N-1 additions. As the matrix size or the number of bits increases, it gets difficult for FPGA to perform all these computations in one clock cycle. We thus pipeline the matrix-vector multiplication such that a new vector-vector multiplication


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 $a_{(1,1:8)}$ x $y_{(1:8,1)}$



(b) Pipeline architecture to compute A x y

Figure 5.7 Pipeline architecture of System Dynamics component for an 8^{th} order system

is instantiated every cycle. The number of these instantiations is equal to the number of rows z in the matrix. Figure 5.7(a) shows different stages in the vector-vector multiplication of $a_{(1,1:8)}$ x $y_{(1:8,1)}$ for an 8^{th} order system. Figure 5.7(b) shows the number of cycles taken by the pipeline architecture to compute A x y. The matrix-vector multiplication for B x u observes the similar architecture and can be implemented in parallel with A x y. However, to improve the timing characteristics of the design we serialize them.

In a six stage pipelined architecture for system dynamics component, the first cycle is used to compute a vector-vector multiplication. In the next cycle the required number of bits from the product are saved in the zeroth vector of a 2D array. In the third cycle, 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 cycle, we add these four data values and save the result in the second vector location followed by addition of the two data values in the fifth cycle. An additional cycle is consumed to pipeline the result. For an 8th order system, each of these stages are implemented for the eight vector-vector multiplications and thus takes 13 cycles to compute A x y, as shown in Figure 5.7. The multiplication of B x u also goes through the same stages with an extra stage to add the result of two multiplications which makes the cycle count as 14 for the second multiplication. The system output takes 12 cycles to compute intermediate $\dot{y}_l$ states and RK4 controller which controls the state transition takes another 13 cycles to change states between the two components. Since there are 4 stages, the total time taken is 4x14 + 4x13 + 4x12 + 13 = 169 clock cycles.

Another variation of this implementation is when system input u is scalar instead of a vector quantity. In this case B matrix is a vector of size N x 1 and B x u is reduced to a vector-vector multiplication. This optimization, first, allows the parallel implementation of the two multiplications, A x y and B x u and, second reduces the number of state transitions. RK4 controller which controls the state transition now takes 9 cycles instead of 14 because the two multiplications can be done in parallel. The total time taken is thus reduced to 4x14 + 4x12 + 9=113 clock cycles.

5.4 Design Layout

From the CPU-based simulator it is clear that the input of one unit is dependent on the output of previous unit. The sequential flow of information thus requires that a high-level architecture of the CPU-based simulator is maintained in the FPGA-based simulator. However, difference in the architecture of each unit changes the way the data-flow is maintained in the latter. The basic components that make up these units (will be referred to as component here after) have already been discussed in Chapter 4 and we use these components to explain the architecture and data-flow in the FPGA-based simulator shown in Figure. 5.8 for the vehicle system.

For the first implementation strategy the FPGA-based simulator of the vehicle system consists of four major components: valve opening area, orifice flow rate, state space solver for valve model and state space solver for vehicle model. Because of resource constraints on a single FPGA we use both the FPGAs (FPGA A and FPGA B) provided by XtremeData platform [51] and the shaded blocks in Figure 5.8 shows the partition of the components. We will soon discuss the criteria behind this distribution in detail. FPGA A implements the valve opening area of the steering valve and RK4 integrator of the vehicle model whereas FPGA B implements the orifice flow rate and RK4 integrator of steering valve model. FPGA A is capable of exchanging data with the host and the FPGA B whereas FPGA B can exchange data only with FPGA A. The wrapper that instantiates the components on the two FPGAs thus implements a finite state machine (FSM) which synchronizes the data-flow between the two FPGAs and with the host. Since the interface connecting the two FPGAs and the FPGA A with the host is 256 bit wide the FSMs also pad the outgoing data with zeros and extract the required I+F bits from the incoming data.

The wrapper on FPGA A scans the most significant 32 bits of the data packet being sent by the host. These bits categorize the data in the first 128 bits of the packet as either of the following: data to reset the simulator, data to initialize the state of the system and data representing the system input, received in that order. When the system input is received the valve opening area reads I+F bits and starts the computation. When the done signal from this component goes high, marking the validity of the data, the output vector is padded with zeros to make each value of maximum allowable length i.e. 128 bit wide. Since the bus is 256 bit wide and there are four 128 bit values in the vector, they are sent across in two packets. The component then stalls until it receives new state information from the state space solver of the steering valve model. Recall from equation (5.4) the computation of $A(\theta)^{*}$ C_d and *sqrt* can be done independently of each other. While the valve opening area is busy computing the former, system input is also sent to the orifice flow rate to compute the latter. It also stalls after computing the *sqrt* and waits for a valid output from the valve opening area. The wrapper

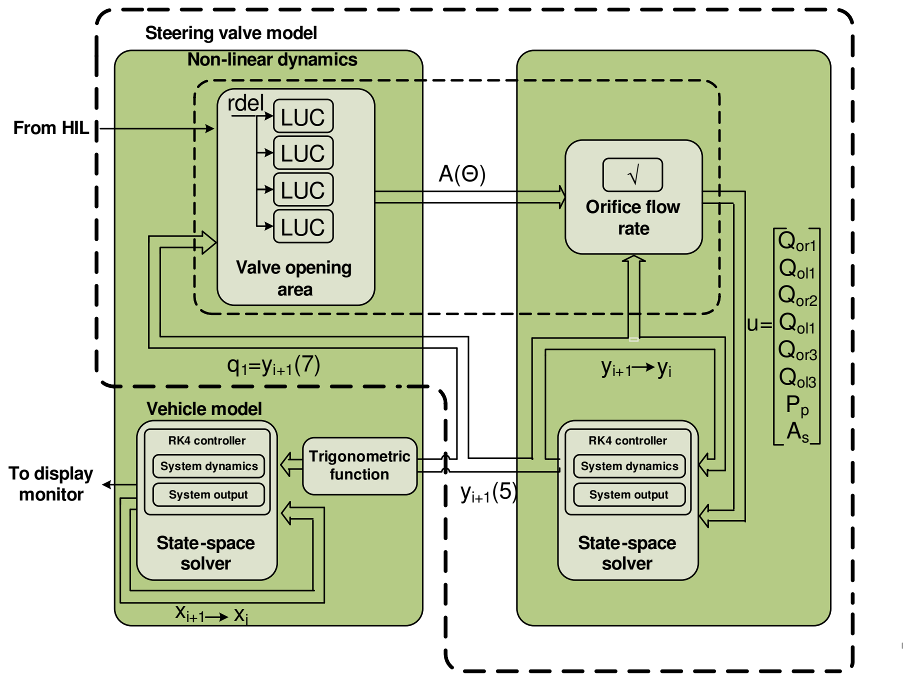


Figure 5.8 Design Layout of FPGA-based simulator for the Vehicle System

on FPGA B scans the incoming interface from the FPGA A. A valid signal on this interface indicates that a valid data is available on the input data bus. The valid signal remains high for two clock cycles, since two 256 bits packet are sent. The output from orifice flow rate is a vector of size eight, each element being I+F bits wide and a done signal to mark the validity of the data on its output bus. Its output forms the system input for the state space solver of the steering valve model which starts computation when the done signal goes high. The output from the state space solver is fed back to both the orifice flow rate and valve opening area. The former reads all the states of the system whereas the latter needs only the state variable q_1, Figure 5.2(a).

The vehicle model starts execution, only when steering valve model has completed. However, owing to difference in their time steps there is no one-to-one relation in the number of times they are executed. When the steering valve model completes $\frac{S}{h_{valve}}$ iterations, vehicle model runs for $\frac{S}{h_{vehicle}}$ iterations. Thus, when state space solver of the steering valve completes the execution, the wrapper on FPGA B checks if $\frac{S}{h_{valve}}$ iterations have completed. If no, then the FSM sends the state variable q_1 and x (piston displacement), used to compute system input for state space solver of the vehicle model, to FPGA A and waits for next valid output from the valve opening area. If yes, 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, if $\frac{S}{h_{valve}}$ iterations have completed, 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 $\frac{S}{h_{vehicle}}$ iterations. In parallel, the counter for $\frac{S}{h_{valve}}$ iterations is reset to 0 and valve opening area component is set to wait for a new system input from the host. Recall from section 3.4 in Chapter 3, the system input does not change until simulation has run for $\frac{S}{h_{valve}}$ and $\frac{S}{h_{vehicle}}$ iterations. And the new system input will be received only when FPGA A has sent the simulation output from the vehicle model back to the software.

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

The orifice flow rate component instantiates a square root component, input to which is the difference between various states of the valve determined by the state space solver. The difference is computed in a pipelined manner such that a new input is 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 state space solver which implements the RK4 integrator has already been discussed in previous section. The only difference is in the system input for the two models and the number of bits used for fixed-point representation. The system input u for the steering valve model is a vector computed by the orifice flow rate component given in Figure 5.2(b). The system input u for the vehicle model is a scalar quantity, computed by applying sinusoidal inverse function to the piston displacement x. Since we compute the inverse function, the input to the trigonometric component is normalized to a value between 0 and 1. However, for the vehicle system in consideration, the piston displacement is further limited to the range $\pm.254$ 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 on MATLAB while running the steering valve model on the FPGA. The resource utilization for this implementation was close to 100% but we were able to fit the design on a single FPGA. The implementation of the design remains the same since the individual components are same however instead of using dual FPGAs we used just one and avoided 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, runs for 20ms.

5.5 Data Flow and Cycle Estimate

The valve opening area uses initial two cycles to compute A_m, equation (5.3) and *rdel*, equation (5.2) respectively. The division operator required for computation of A_m using equation (5.3) is implemented with an inverse operation. Since I_g remains constant throughout the simulation, we save its inverse in the parameters file while generating the VHDL design. *rdel* is computed in the second cycle by taking $2'$s complement of A_m and adding the result to A_s. The look-up curve takes five cycles to generate the output and an additional cycle to obtain the product with Cd followed by a cycle to pipeline the output. Thus total of 9 cycles are used to obtain the output from a single look-up curve component. Four look-up curve components for four valves are instantiated in a pipelined manner for which the output is available in twelve cycles. The output, $A(\Theta)$*Cd for each valve is saved in vector $\vec{A}(\Theta)$. An additional cycle is consumed in 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 equation(4.2). Since the product A(Θ)*Cd has already been computed in valve opening area, orifice flow rate handles two functions of the equation (5.4) (1)square root (2)multiplication of output 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 $2'$s complement of p_f to p_i. The absolute of the difference is obtained by taking its $2'$s complement only if the difference negative i.e the MSB is 1. To replicate the sign of the difference in the final output from this component we pipeline this sign bit to be used in latter stage. The square root of the difference is initiated in the second cycle and the component takes 12 cycles to return the square root. Thus, a single square root computation takes 14 cycles and a pipelined architecture for six such computations (for six flow rates) takes 19 cycles.

The valve opening area and orifice flow rate are being executed in parallel. After receiving the new state information from the RK4 the orifice flow rate starts the computation immediately and stalls after 19 cycles until it receives the latest value of $\vec{A}(\Theta)$ from valve opening area. So when the valve opening area starts the computation, the orifice flow rate is already in a state waiting for input from the valve opening area unit. The time taken to complete one iteration of steering valve thus includes 13 cycles for valve opening area, followed by the 15 cycle delay associated with sending two 256 bits packet to FPGA B. Orifice flow rate takes 8 cycles to implement the second function. One cycle to read the valve area,six cycles to compute six flow rate values and one cycle to set up the input vector for the state space solver. The state space solver for the valve model takes 169 cycles, as explained in section 2.1 and further 13 cycles are taken in sending two state output to the FPGA A. Thus, each iteration of the steering valve model is completed in 226 cycles. Further on, 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, as explained in section 2.1.

5.6 XD2000i Architecture

XtremeData's XD2000i development system [51] is chosen to implement the design of FPGA-based simulator. The system consists of a development PC with Xeon®dual-processor system, linux CentOS operating system and an XD2000i FPGA in-socket accelerator that plugs directly into the processor system. The XD2000i module features three Stratix III EP3SE260F1152C3 Altera FPGAs, one bridge and two application (FPGA A and FPGA B), each with 254,400 logic elements (101,760 ALMs), a 1067M front-side bus (FSB) interface that provides a bandwidth of 8.5 GB/s, two QDRII+ 350MHz SRAM each of 8MB, connected with two application FPGAs through an interface that provides a bandwidth of 2.8GB/s. The Bridge FPGA is dedicated to implement the FSB protocol that connects the bridge FPGA to the Northbridge on one side and to the two application FPGAs on the other side. It is not modifiable by the user and only the application FPGAs are used for implementation. The bus connecting the bridge and the two application FPGAs is a 64-bit wide unidirectional bus running at 200MHz. The data between the two application FPGAs is transferred through a 256-bit wide bus running at 100MHz.

The software partition runs on Xeon processor and implements the software controller that uses blocking send and receives functions to communicate with the application FPGAs. The blocking functionality implies that once a send function has been executed i.e data has been sent to either of the FPGAs, the receive function cannot be executed. Alternatively, for every send command to the FPGA the next send cannot be executed until the response to the previous send has been received. An additional requirement for this communication process is that, before the controller can initiate the transfer it should have the information about the number of bits that should be sent and the number of bits that are expected from the FPGA. It then prepares a send and a receive buffer of the required size. Any mismatch between the size of the buffer and the number of bits sent or received will cause the controller and the hardware to stall.

Figure.5.9 describes partitioning of the design across two FPGAs. The partitioning algorithm is governed by two factors. First, implementation on the XtremeData platform requires

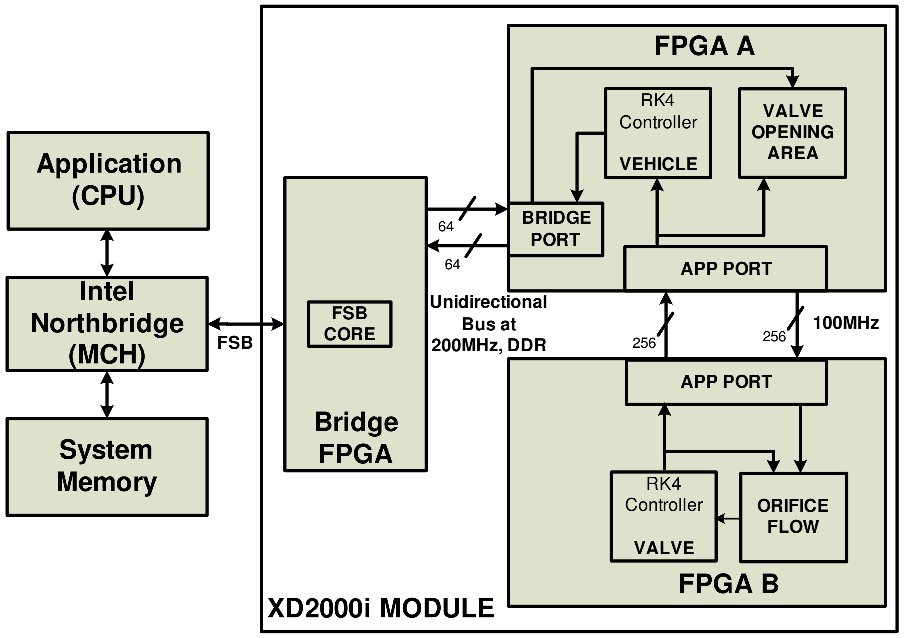


Figure 5.9 Design Partitioning on XD2000i Architecture

that the same application port is used to exchange data between the application FPGA and the software controller. Hence, the valve opening area component, which receives the system input from the software controller, and the state space solver of vehicle model, which generates the simulation output i.e. the new state of the vehicle, are implemented on the same FPGA. Second, for the bit combination of M=96, F=47 for the steering valve model and M=56, F=46 for the vehicle model, Table 5.3 shows the percentage resource utilization of different components. Resource utilization of FPGA A is thus estimated to be 29+11+5=45% and has the capacity to accommodate more components. However, if orifice flow rate (which uses square root core) and state space solver for steering valve model are implemented on separate FPGAs, there will be a huge communication delay involved in sending data of eight states after every iteration from one FPGA to another. To counter this problem and also to efficiently utilize resources of both the FPGAs, we implement orifice flow rate and state-space solver on FPGA B with an estimated resource utilization of 38+50=88%. Recall that the state of the valve model which is used to compute the system input for the vehicle model represents the piston displacement. To obtain the equivalent angular displacement trigonometric, function which implements the

Table 5.3 Resource usage by different components

Component	Percentage Resource Usage
RK4 Valve	38%
RK4 Vehicle	29%
Look-up Curve	11%
Square Root	50%
Trigonometric	5%

inverse of sin, is used before the input is fed to the vehicle model.

To run the RTS of the vehicle system using FPGA-based simulator h_{valve} is set to 10^{-6} s and $h_{vehicle}$ is set to 2^{-3} s. The software controller sends the steering wheel angle, A_s to FPGA A every 20 ms. This includes the communication delay over FSB to send the system input and receive the simulation output and also the time required to compute new state of the vehicle. After sending the output, the simulator stalls until it receives a new system input from the controller. Once the controller receives the simulation output it stalls until 20 ms have completed before it can send a new system input.

5.7 Simulation and Synthesis Results

The maximum frequency supported by XtremeData (XD2000i) platform is 100MHz. However the design was only able to meet the timing constraints at clock frequency of 55 MHz (18.18ns time period). We first compared the time taken by a single iteration on FPGA, using Modelsim simulator, with the MATLAB based model running on an Intel Core2 Quad 2.83 GHz processor. The data exchange between the two FPGAs uses FIFOs. The only instance when the data is written by FPGA A to its exit FIFO, is when it has to send 512 bits containing the four valve opening area results to FPGA B. It is important to note here that FPGA A does not write continuously to this FIFO and by the time the second set of 512 bits is written, the first one has already been read, so we do not have scenario where FIFO full signal will go high. Similarly, FPGA B writes to its exit FIFO when it has to send 256 bits containing 2 states of the valve model to FPGA A. The inflow to the FIFO is 256 bits per iteration and by the time next 256 bits are written the first one has already been used. We thus assume that the delay associated in sending data across the FPGA is closely simulated in Modelsim simulation to the actual delay on the hardware.

The MATLAB based model takes 13μs to complete one iteration of steering valve and one iteration of vehicle. The Modelsim shows that one iteration of steering valve model takes 4.1μs, which includes a delay of .27μs (15 cycles) required to send 512 bits of data from FPGA A to FPGA B. A further delay of .234μs (13 cycles) is observed, while sending the 256 bit output of the valve model to FPGA B. The vehicle model generates output in 2.214μs (123 cycles). Apart from RK4 component for the vehicle model which takes 114 cycles and 7 cycles are consumed by the trigonometric function, a cycle is used to obtain the input for the trigonometric functions by division of linear piston displacement, x with the length of the arm. A cycle is consumed to complement the output of the trigonometric function. Thus, the total time estimated from Modelsim to compute a single iteration of the steering valve and each iteration of the vehicle model is 4.1+.27+.234+2.214=6.818μs. This comparison does show a speedup of 2 times over the MATLAB implementation. However the actual simulation allows steering valve model to run for 20 ms followed by vehicle model simulation for 20 ms. In Modelsim, the time required to generate final output of vehicle system simulation for 20 ms is computed as 9.21ms which shows a speedup of 2$\times$ over the required time of 20 ms.

The above comparison does not include the delay in sending data over the FSB interface. We now compare the time taken by the hardwar/software design set-up which includes the time required to send the system input to FPGA, run the simulation on FPGA and receives the simulation results. The timer starts when the software controller sends the system input to FPGA and stops after the simulation results have been returned to the controller. This time is computed as 10ms, which shows a speedup of 2$\times$ over 20ms. Since the vehicle system simulation running on FPGA expects a new system input every 20ms, we stall the software controller for 10 ms before sending the next system input.

For the second implementation we were able to implement the steering valve model to run at 62.5MHz on a single FPGA. The total time to run the simulation for 20ms was computed

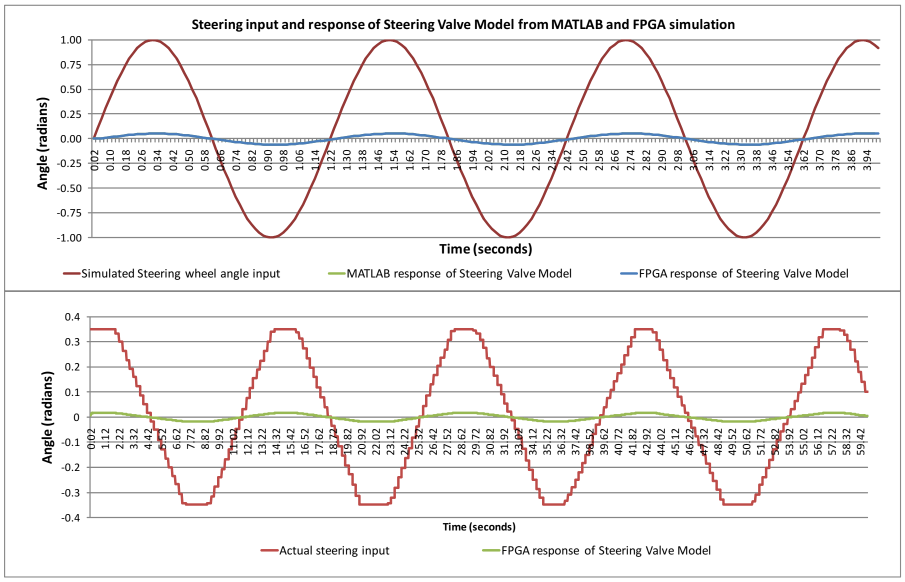


Figure 5.10 Response of the Steering Valve model to simualted and actual Steering input

as 7ms excluding the communication delay of 1 ms and we stalled the software for 12ms. The simulation results were sent to the vehicle model on the FPGA which ran the vehicle model for 20ms and computed the position co-ordinates to be sent to the VR display monitor. For second implementation since we used 3 different architectures to drive the simulation, the simulation was slightly lagging because of the socket delay but the computation of the results is not lagging behind. If the computation is lagging behind then over a period of time we would observe a delayed vehicle movement in response to the change in the steering input because of aggregated error in computation of the results. To show that the FPGA is generating correct results for the steering valve model which further drives the vehicle model, Figure 5.10 compares the piston displacement angle from the steering valve model in response to a simulated steering input of a sinusoidal wave and the actual steering wheel input. We also plot the piston displacement angle for same set of input from the MATLAB version of the model which overlap with the output from the FPGA. The socket communication on an average takes 16μs to send the data across and considering the time taken by FPGA to generate the output the socket delay is a constant delay which does not increases over a period of time.

CHAPTER 6. RELATED WORK AND FUTURE RESEARCH

6.1 Related Work

Since the existence of the FPGAs in mid 1980s they have been used in various fields for prototyping, acceleration and reconfiguration for different computations. [15] outlines the benefits of FPGA implementation in various fields and the advantages of such reprogrammable systems. Initially, the FPGAs were either used to emulate the ASIC targeted applications to test the design before the production of custom hardware [30] or to accelerate computation intensive applications which would otherwise show poor performance with software implementation. For example, temporal pattern and speech recognition using 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 the matching becomes computation intensive. The parallel architecture of FPGAs enhance the search process by parallel execution of the independent steps and thus provide an appropriate platform for such applications [38]. The other computation intensive applications which have successfully explored the parallel architecture of FPGAs are graph problems such as Hamiltonian cycle [40], mathematical methods such as finite-difference time-domain [11] and Jacobi iteration [29], communications decoding algorithms [32].

Another area where FPGAs have played a significant role is in the performance improvement of algorithms for Molecular dynamics (MD). MD simulates the motion and interaction between atoms or molecules based on different forces acting between these particles. It is the computation of these forces that has become enormously expensive to be performed on a single processor. [2] in 2004 was the first work published which tested the feasibility of MD using FPGAs. [20], [39], [14] further demonstrate the usefulness and performance improvement of

FPGA-based MD simulations. Yet another field of application is Bioinformatics and the earliest use of hardware acceleration for biological sequence comparison was in 1998 via dedicated hardware, SAMBA (Systolic Accelerator for Molecular Biological Applications) accelerator [21]. [12] achieved a speedup of 200 times over the conventional desktop implementation for protien sequence alignment and [3] shows a speedup of 383 times for the multiple DNA sequence alignment by implementing computation intensive part of comparison algorithm on FPGA. With the enormous progress made in the field of FPGA-based acceleration, the financial modelling methods are also being experimented for FPGA implementation. [53], [45] and [49] demonstrate the speedup of upto 80 times for the computation intensive Monte Carlo simulation algorithm. [17] explores the parallel architecture of FPGA for portfolio management. Guassian distribution models which are used to model correlation between different entities such as finding correlation between portfolios containing hundreds of assets used FPGA-based implementation for this model and achieved a speedup of 33 times over CPU-based implementation [46].

The usage of FPGAs has recently grown in the accelration of real-time simulation of systems as mentioned in Chapter 1. [9] and [10] achieved the real-time simulation for permanent magnet synchronous motors using RT-LAB real-time simulation platform and auto generation of the hardware blocks using Xilinx Simulink Generator (XSG). Apparently not much work has been done using auto-generation of the hardware design. The progress has been made towards manually desiging the hardware design to simulate high frequency power system model using FGPAs to study the dynamic behvaior of large systems. [22] explored hardware-software codesign approach for dual time step real-time simulation of power systems. For high frequency transient phenomenons in power systems such as power electronic switching it is possible to simulate such systems using FPGA as the main computation core because of the reduced execution time offered while keeping the systems with larger time step on CPU. Recently [6] proposed an FPGA-based real-time electromagnetic-transient program simulator which is capable of simulating systems with a time-step of 12μs when the acceptable time step for transient simulation is 50μs. FPGAs thus offer a promising platform for the simulation of fast transients in power systems.

Not much work has been done in the field of acceleration of vehicle system dynamics using FPGAs. Recently, [55] demonstrated real-time simulation of railway-vehicle dynamics using FPGA-based accelerator. A fast MATLAB/SIMULINK implementation of railway-vehicle with a time step of 1 ms completes each step in 21.5 ms whereas with the FPGA the execution takes .625 ms. However, the time does not include the communication delay involved due to distributed simulation architecture. In this thesis, we share the advantage of improved performance with the above mentioned work in addition to the successful real-time simulation with the distributed architecture which includes HIL and VR-display. We propose a framework to automatically generate the hardware design with sufficient accuracy that would fit on the hardware. This paper is the initial step towards the final goal of developing FPGA-based simulators for vehicle simulation models with driver-in-loop.

6.2 Conclusion and Future Research

In this thesis, we present a method to improve the simulation time of vehicle systems, to meet the real-time constraints using hardware based implementation of the mathematical models. We present the methodology adopted to implement these models, for different orders and estimate the resource usage of the hardware design beforehand to make intelligent decisions about the implementation strategy. We applied our methodology to an 8^{th} order steering valve and vehicle model. The system was successfully implemented on the hardware with a speedup of $2.2\times$ for the overall simulation process. During the process, we designed hardware components, that can be further used for implementation of other models. This work forms the basis for the next step of research in this direction which will focus on developing partitioning algorithms to provide different architectures for implementing the models across multiple hardware platforms along with the software integration. In our work we considered the implementation across dual FPGAs, the work can be formalized to consider any number of hardware platforms and obtain the best hardware/software and hardware/hardware partitions.

BIBLIOGRAPHY

[1] S. Alles, C. Swick, S. Mahmud, and F. Lin. Real time hardware-in-the-loop vehicle simulation. In *Proceedings of Instrumentation and Measurement Technology Conference (IMTC)*, pages 159–164, 12-14 1992.

[2] N. Azizi, I. Kuon, A. Egier, A. Darabiha, and P. Chow. Reconfigurable molecular dynamics simulator. In *Proceedings of Field-Programmable Custom Computing Machines (FCCM)*, pages 197–206, April 2004.

[3] Azzedine Boukerche, Jan Correa, Alba de Melo, Ricardo Jacobi, and Adson Rocha. An FPGA-based accelerator for multiple biological sequence alignment with DIALIGN. In *High Performance Computing (HiPC)*, volume 4873 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin / Heidelberg, 2007.

[4] Gregorio Cappuccino, Pasquale Corsonello, and Giuseppe Cocorullo. High performance vlsi modules for division and square root. *Microprocessors and Microsystems*, 22(5):239 – 246, 1998.

[5] Chapra and Canale. *Numerical Methods for Engineers.* Tata McGraw-Hill, 5th edition, 2006.

[6] Yuan Chen and V. Dinavahi. FPGA-based real-time EMTP. *IEEE Transactions on Power Delivery*, 24(2):892 –902, Apr. 2009.

[7] Marco Cipellia, Werner Schiehlenb, and Federico Cheli. Driver-in-the-loop simulations with parametric car models. In *International Journal of Vehicle Mechanics and Mobility*, volume 46, pages 33–48, 2008.

[8] O. Devaux, L. Levacher, and O. Huet. An advanced and powerful real-time digital transient network analyser. *IEEE Transactions on Power Delivery*, 13(2):421 –426, Apr. 1998.

[9] C. Dufour, S. Abourida, and 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*, pages 2591–2596, Jul. 2006.

[10] C. Dufour, J. Belanger, S. Abourida, and V. Lapointe. FPGA-based real-time simulation of finite-element analysis permanent magnet synchronous machine drives. In *Proceedings of Power Electronics Specialists Conference (PESC)*, pages 909–915, Jun. 2007.

[11] J.P. Durbano and F.E. Ortiz. FPGA-based acceleration of the 3d finite-difference time-domain method. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 156–163, Apr. 2004.

[12] Stefan Dydel. Large scale protein sequence alignment using FPGA reprogrammable logic devices. In *Proceedings of International Conference on Field Programmable Logic and Application (FPL)*, pages 23–32, 2004.

[13] D.D. Gajski and F. Vahid. Specification and design of embedded hardware-software systems. *IEEE Design and Test of Computers*, 12(1):53–67, 1995.

[14] Yongfeng Gu, Tom VanCourt Martin, and C. Herbordt. FPGA-based multigrid computation for molecular dynamics simulations. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 117–126, 2007.

[15] S. Hauck. The roles of FPGAs in reprogrammable systems. *Proceedings of the IEEE*, 86(4):615 –638, Apr. 1998.

[16] Scott Hauk and Andre Dehon. *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation*. Morgan Kaufmann, 2008.

[17] Ali Irturk, Bridget Benson, Nikolay Laptev, and Ryan Kastner. FPGA acceleration of

mean variance framework for optimal asset allocation. In *Proceedings of Workshop on High Performance Computational Finance (WHPCF)*, pages 1–8, 2008.

[18] Manoj 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)*, May 1995.

[19] Manoj Karkee and Brian L Steward. Open and closed loop system characteristics of a tractor and an implement dynamic model. In *American Society of Agricultural and Biological Engineers (ASABE)*, 2008.

[20] Jun-Ho Kim, Chun-Sik Park, Sang-Gyum Kim, and Jung-Ha Kim. Improved interpolation and system integration for FPGA-based molecular dynamics simulations. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, pages 21–28, 2006.

[21] Dominique Lavenier. Speeding up genome computations with a systolic accelerator. `http://www.irisa.fr/symbiose/lavenier/Publications/Lav98ja.pdf`, 1998.

[22] P. Le-Huy, S. Guerette, L.A. Dessaint, and Hoang Le-Huy. Dual-step real-time simulation of power electronic converters using an FPGA. In *Proceedings of IEEE International Symposium on Industrial Electronics*, pages 1548–1553, 2006.

[23] R. C. Lee and F. B. Cox. A high-speed analog-digital computer for simulation. *IEEE Transactions on Electronic Computers (EC)*, 8(2):186–197, June 1959.

[24] M. Lerotic, Su-Lin Lee, J. Keegan, and Guang-Zhong Yang. Image constrained finite element modelling for real-time surgical simulation and guidance. In *Proceedings of Biomedical Imaging: From Nano to Macro*, pages 1063–1066, June 2009.

[25] Michael D. Letherwood and David D. Gunter. Ground vehicle modeling and simulation of military vehicles using high performance computing. *Parallel Computing*, 27(1-2):109–140, 2001.

[26] J. Maroto, E. Delso, J. Felez, and J.M. Cabanellas. Real-time traffic simulation with a microscopic model. *IEEE Transactions on Intelligent Transportation Systems*, 7(4):513–527, Dec. 2006.

[27] Giovanni De Micheli and Rajesh K. Gupta. Hardware/software co-design. *IEEE MICRO*, 85:349–365, 1997.

[28] P. Montuschi and P.M. Mezzalama. Survey of square rooting algorithms. *IEE Proceedings E on Computers and Digital Techniques*, 137(1):31–40, Jan. 1990.

[29] G.R. Morris and V.K. Prasanna. An FPGA-based floating-point jacobi iterative solver. In *Proceedings of International Symposium on Parallel Architectures,Algorithms and Networks (ISPAN)*, page 8, Dec. 2005.

[30] S. Note, P. van Lierop, and J. van Ginderdeuren. Rapid prototyping of DSP systems: requirements and solutions. In *Proceedings of IEEE International Workshop on Rapid System Prototyping*, pages 88 –96, Jun. 1995.

[31] Lok-Fu Pak, M.O. Faruque, Xin Nie, and V. Dinavahi. A versatile cluster-based real-time digital simulator for power engineering research. *IEEE Transactions on Power Systems*, 21(2):455–465, May 2006.

[32] B. Pandita and S.K. Roy. Design and implementation of a viterbi decoder using FPGAs. In *Proceedings of the International Conference on VLSI Design*, pages 611–614, Jan. 1999.

[33] Andrew Percey. Advantages of the Virtex-5 FPGA 6-Input LUT Architecture. `http://www.xilinx.com/support/documentation/white_papers/wp284.pdf`, December 2007. Available online (6 pages).

[34] Laura R. Ray. Nonlinear estimation of vehicle state and tire forces. In *Proceedings of American Control Conference*, pages 526 –530, 24-26 1992.

[35] A.P. Sage and S.L. Smith. Real-time digital simulation for systems control. *Proceedings of the IEEE*, 54(12):1802–1812, Dec. 1966.

[36] D.J. Sandoz and B.H. Swanick. Real-time hybrid simulation of an adaptive-control technique. *Proceedings of the Institution of Electrical Engineers*, 117(11):2165 –2173, Nov. 1970.

[37] Michael W Sayers. Vehicle models for RTS applications. In *International Journal of Vehicle Mechanics and Mobility*, volume 32, pages 421–438, 1999.

[38] H. Schmit and D. Thomas. Hidden markov modeling and fuzzy controllers in FPGAs. In *Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines*, pages 214–221, Apr. 1995.

[39] Ronald Scrofano, Maya B. Gokhale, Frans Trouw, and Viktor 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)*, pages 23–34, 2006.

[40] M. Serra and K. Kent. Using FPGAs to solve the hamiltonian cycle problem. In *Proceedings of the International Symposium on Circuits and Systems (ISCAS)*, pages 228–231, May. 2003.

[41] P. Soderquist and M. Leeser. An area/performance comparison of subtractive and multiplicative divide/square root implementations. In *Proceedings of the Symposium on Computer Arithmetic*, pages 132–139, 19-21 1995.

[42] Peter Soderquist and Miriam Leeser. Division and square root: Choosing the right implementation. *IEEE Micro*, 17(4):56–66, 1997.

[43] Kulakowski B T, Gardner J F, and Shearer J L. *Dynamic modeling and control of engineering systems*. Cambridge University Press, 2007.

[44] M.J. Tavernini, B.A. Niemoeller, and P.T. 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)*, pages 890–895, 710 2009.

[45] David B. Thomas, Jacob A. Bower, and Wayne Luk. Automatic generation and optimisation of reconfigurable financial monte-carlo simulations. In *Proceedings of International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 168–173, 2007.

[46] David B. Thomas and Wayne Luk. Sampling from the multivariate gaussian distribution using reconfigurable hardware. In *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 3–12, 2007.

[47] D.E. Thomas, J.K. Adams, and H. Schmit. A model and methodology for hardware-software codesign. *IEEE Design and Test of Computers*, 10(3):6–15, Sep. 1993.

[48] X. Wang and R.M. Mathur. Real-time digital simulator of the electromagnetic transients of transmission lines with frequency dependence. *IEEE Transactions on Power Delivery*, 4(4):2249 –2255, Oct 1989.

[49] N.A. Woods and T. VanCourt. FPGA acceleration of quasi-monte carlo in finance. In *Proceedings of International Conference on Field Programmable Logic and Applications (FPL)*, pages 335–340, 8-10 2008.

[50] Xu Xiaobo, Zheng Kangfeng, Yang Yixian, and Xu 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)*, pages 237–241, 18-20 2009.

[51] XtremeData. Xd2000i^{TM}development system. `http://www.xtremedata.com/products/accelerators/development-systems/xd1000dsi-ds`.

[52] Randy Yates. Fixed-point arithmetic: An introduction. `www.digitalsignallabs.com/fp.pdf`, 2007.

[53] G.L. Zhang, P.H.W. Leong, C.H. Ho, K.H. Tsoi, C.C.C. Cheung, D.-U. Lee, R.C.C. Cheung, and W. Luk. Reconfigurable acceleration for monte carlo based financial simulation. In *Proceedings of International Conference on Field-Programmable Technology (FPL)*, pages 215–222, Dec. 2005.

[54] Shupeng Zheng, Shutao Zheng, Jingfeng He, and Junwei Han. An optimized distributed real-time simulation framework for high fidelity flight simulator research. In *Proceedings of International Conference on Information and Automation (ICIA)*, pages 1597–1601, 22-24 2009.

[55] Y.J. Zhou, T.X. Mei, and S. Freear. Field programmable gate array implementation of wheel-rail contact laws. *Control Theory Applications, IET*, 4(2):303 –313, Feb. 2010.