

Rowan University

Rowan Digital Works

Theses and Dissertations

6-17-2024

HARDWARE ACCELERATION OF NUMERICAL METHODS FOR SOLVING ORDINARY DIFFERENTIAL EQUATIONS

Soham Bhattacharya
Rowan University

Follow this and additional works at: <https://rdw.rowan.edu/etd>



Part of the [Applied Mathematics Commons](#), and the [Computer Engineering Commons](#)

Recommended Citation

Bhattacharya, Soham, "HARDWARE ACCELERATION OF NUMERICAL METHODS FOR SOLVING ORDINARY DIFFERENTIAL EQUATIONS" (2024). *Theses and Dissertations*. 3245.
<https://rdw.rowan.edu/etd/3245>

This Thesis is brought to you for free and open access by Rowan Digital Works. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Rowan Digital Works. For more information, please contact graduateresearch@rowan.edu.

**HARDWARE ACCELERATION OF NUMERICAL METHODS FOR
SOLVING ORDINARY DIFFERENTIAL EQUATIONS**

by
Soham Bhattacharya

A Thesis

Submitted to the
Department of Electrical and Computer Engineering
College of College of Engineering
In partial fulfillment of the requirement
For the degree of
Master of Science in Electrical and Computer Engineering
at
Rowan University
May 28, 2024

Thesis Chair: Dwaipayan Chakraborty, Ph.D., Assistant Professor, Department of
Electrical and Computer Engineering, Rowan University.

Committee Members:

Huaxia Wang, Ph.D., Assistant Professor, Department of Electrical and Computer
Engineering, Rowan University
Thanh Nguyen, Ph.D., Associate Professor, College of Science and Mathematics,
Rowan University.

© 2024 Soham Bhattacharya

Dedications

I would like to dedicate this thesis to my parents. Without their support, this would never have been possible.

Acknowledgements

Writing this thesis has constituted a significant portion of my academic experience. This journey would not have been possible without the constant backing and motivation from numerous individuals. First, I would like to express my gratitude to Dr. Dwaipayan Chakraborty for giving valuable insights and guidance on my research, and for allowing me to do research at Rowan University. The skills and knowledge that I have gained at Rowan University are things that I will carry with me on my next professional journey. I would also like to thank all my committee members, Dr. Thanh Nguyen and Dr. Huaxia Wang for giving their valuable time for my research. My parents, Mr. Joyabrata Bhattacharya and Mrs. Keya Bhattacharya deserve my deepest gratitude for their immense support throughout this process. Without them, I wouldn't have been able to step foot into the USA and accomplish my dreams. They made sacrifices over the course of their entire lives that enabled me to pursue a foreign degree and fulfill even my wildest aspirations. Next, I want to thank my partner, Ms. Srija Samaddar for her continuous mental support throughout this whole process when I am not in a stable state to fight. I truly appreciate the time and effort they have put into supporting me through this process. In addition to them, I would like to express my gratitude towards my close friends and brothers, Hare and Anirban, for their significant contribution of mental help whenever it is needed during this entire journey. I would also like to thank all my family members, roommates, fellow colleagues, friends and everyone associated with this degree.

Abstract

Soham Bhattacharya

HARDWARE ACCELERATION OF NUMERICAL METHODS FOR SOLVING ORDINARY DIFFERENTIAL EQUATIONS

2023-2024

Dwaipayan Chakraborty, Ph.D.

Master of Science in Electrical and Computer Engineering

Along with the advancement in technology, the role of hardware accelerators is increasing consistently, delivering advancements in scientific simulations and data analysis in scientific computing, signal processing tasks in communication systems, matrix operations, and neural network computations in artificial intelligence and machine learning models. On the other hand, several high-speed computer applications in this era of high-performance computing often depend on ordinary differential equations (ODEs); however, their nonlinear nature can present a challenge to obtaining analytic solutions. Consequently, numerical approaches prove effective in delivering only approximate solutions to these equations. This research discusses the implementation of a customized hardware accelerator for solving an ordinary differential equation (ODE) by utilizing numerical approaches while evaluating several performance metrics, including on-chip power consumption, FPGA hardware resources, and timing summary. The third-party vendor AXI4 stream Xilinx single-precision floating-point IP support has been used to develop the accelerator for solving the ordinary differential equation using those methods. The accelerator will determine the iteration approximation result of the ODE using those methods. The entire work uses VHDL hardware description language and the Xilinx Vivado Design Suite and has been deployed on the Zynq-ZC702 FPGA Evaluation Board, along with a design space exploration.

Table of Contents

| | |
|--|----|
| Abstract | v |
| List of Figures | ix |
| List of Tables | xi |
| Chapter 1: Introduction | 1 |
| 1.1 Architecture For General Purpose Processors | 1 |
| 1.2 Challenges Of General Purpose Processors | 2 |
| 1.3 Solution Of Ordinary Differential Equations Using Numerical Methods | 4 |
| 1.4 Overview Of The Proposed Work | 5 |
| Chapter 2: Background | 6 |
| 2.1 Domain-Specific Micro-Architecture | 6 |
| 2.2 RISC-V Micro-Architecture | 7 |
| Chapter 3: Literature Review | 10 |
| 3.1 Commercially Pre-Built Accelerators For Scientific Computing Work- loads | 10 |
| 3.2 Customized, Application-Specific Hardware Accelerators For Scien- tific Computing Workloads | 11 |
| Chapter 4: Motivation | 14 |
| Chapter 5: Methods And Implementation | 16 |
| Chapter 6: Numerical Methods For Solving Ordinary Differential Equations | 27 |
| 6.1 The Euler Method | 27 |

Table of Contents (Continued)

| | | |
|--|--|----|
| 6.2 | The Modified Euler Method | 28 |
| 6.3 | Runge-Kutta Methods | 28 |
| Chapter 7: Micro-Architecture Description..... | | 31 |
| 7.1 | Overview | 31 |
| 7.2 | Micro-Architecture Of The Accelerator | 31 |
| 7.3 | Hardware Implementation Of The Euler And The Modified Euler Methods Modules Within The Accelerator | 38 |
| 7.3.1 | The Euler Method Module | 38 |
| 7.3.2 | The Modified Euler Method Module | 40 |
| 7.4 | An Example Of One Set Of Inputs Being Processed By The Processor For The Euler Method Solver | 42 |
| Chapter 8: Experimental Results | | 45 |
| 8.1 | Power Analysis..... | 45 |
| 8.2 | Hardware Resource Utilization Summary | 50 |
| 8.3 | Timing Summary..... | 52 |
| Chapter 9: Conclusion | | 55 |
| Chapter 10: Future Work..... | | 56 |
| 10.1 | Design-Space Exploration For Runge-Kutta Hardware Accelerator For A System Of Ordinary Differential Equations..... | 56 |
| 10.2 | Hardware Acceleration For Multigrid And Numerical Methods: A Survey | 56 |

Table of Contents (Continued)

References58

List of Figures

| Figure | Page |
|---|------|
| Figure 1. The Von-Neumann Architecture | 2 |
| Figure 2. Overview Of An Architecture Of A Hardware Accelerator (adopted from [16]) | 7 |
| Figure 3. RISC-V Instruction Sets | 9 |
| Figure 4. The Flow Diagram From Application Software To Hardware Representation | 19 |
| Figure 5. A Snippet Of The C code For The Obstacle Detection With Inline Assembly Instructions | 20 |
| Figure 6. Spike Output When The Sensor Value Is 0 | 21 |
| Figure 7. Spike Output When The Sensor Value Is 1. | 21 |
| Figure 8. Disassembly Of The Machine Instructions Into Their Assembly Instructions | 22 |
| Figure 9. A Portion Of The Conversion Of The Assembly Instructions To Json File Format | 23 |
| Figure 10. Generation Of The Verilog Files Of The Processor Design And Its Associated Testbench Using The Chipcron Tool | 24 |
| Figure 11. Simulation Using VVP Tool Through Icarus Verilog | 25 |
| Figure 12. The Internal Representation Of The Accelerator Design In The Form Of A Netlist Using Yosys Synthesizing Tool | 25 |
| Figure 13. Gate-level Synthesis Simulation Results Of The Hardware Accelerator | 26 |
| Figure 14. The Micro-Architecture Design Of The Hardware Accelerator For Solving An Ordinary Differential Equation Using The Euler And The Modified Euler Methods | 33 |
| Figure 15. The Functional Computation Block (FCB) | 39 |

List of Figures (Continued)

| Figure | Page |
|---|------|
| Figure 16. The Micro-Architecture Design Of The Euler Method Module | 39 |
| Figure 17. The Modified Euler Functional Block | 40 |
| Figure 18. The Micro-Architecture Design Of The Modified Euler Method Module | 41 |
| Figure 19. The Flow Diagram Of The Hardware Accelerator For The Euler Method In Solving The ODE | 44 |
| Figure 20. The Percentage Of The Total On-chip Power Consumption By The Euler Method In Solving The ODE In (11), Including The Static And Dynamic Sources | 47 |
| Figure 21. The Percentage Of The Total On-chip Power Consumption By The Modified Euler Method In Solving The ODE In (11), Including The Static And Dynamic Sources | 47 |
| Figure 22. The On-chip Power Consumption Of Second-order, Third-order, And Fourth-order Runge-Kutta Family Methods For Equation (1) In Half-precision Floating-Point Format | 48 |
| Figure 23. The On-Chip Power Consumption Of Second-Order, Third-Order, And Fourth-Order Runge-Kutta Family Methods For Equation (1) In Single-Precision Floating-Point Format | 48 |
| Figure 24. The On-chip Power Consumption Of Second-order, Third-order, And Fourth-order Runge-Kutta Family Methods For Equation (1) In Double-precision Floating-point Format | 49 |

List of Tables

| Table | | Page |
|-----------|---|------|
| Table 1. | The Comprehensive Functional Details Of The Internal Buses For Developing The Hardware Accelerator Using Both Numerical Methods | 34 |
| Table 2. | The Number Of Xilinx Vivado IP Single-Precision Floating-Point IP Support Blocks Essential To Construct The Function In (11) Using The Euler And The Modified Euler Methods | 41 |
| Table 3. | Comparative Table Of On-Chip Power Consumption For Second-Order, Third-Order, And Fourth-Order Of Runge-Kutta Family Equations In Half, Single, And Double Floating-Point Precision Formats | 49 |
| Table 4. | The Hardware Resource Utilization Summary Of The Accelerator Design With The Euler And The Modified Euler Methods In Solving The ODE In (11) | 50 |
| Table 5. | FPGA Hardware Resources For Hardware Accelerator For The RK2 Method In All Floating-Point Precision Formats | 51 |
| Table 6. | FPGA Hardware Resources For Hardware Accelerator For The RK3 Method In All Floating-Point Precision Formats | 51 |
| Table 7. | FPGA Hardware Resources For Hardware Accelerator For The RK4 Method In All Floating-Point Precision Formats | 52 |
| Table 8. | Timing Summary For The Hardware Accelerator For The RK2 Solver | 53 |
| Table 9. | Timing Summary For The Hardware Accelerator For The RK3 Solver | 53 |
| Table 10. | Timing Summary For The Hardware Accelerator For The RK4 Solver | 54 |

Chapter 1

Introduction

High-performance computing is the ability to perform complex computations at very high speeds. High-performance computing includes parallel computing, cluster computing, and grid computing. In this realm of high-performance computing, there has been an assumption amongst programmers that, with the new generation of microprocessors, their software will run significantly faster and more smoothly. Recent research in the field, however, raised a question on how we can optimize and enhance the software utilization in these new architectures.

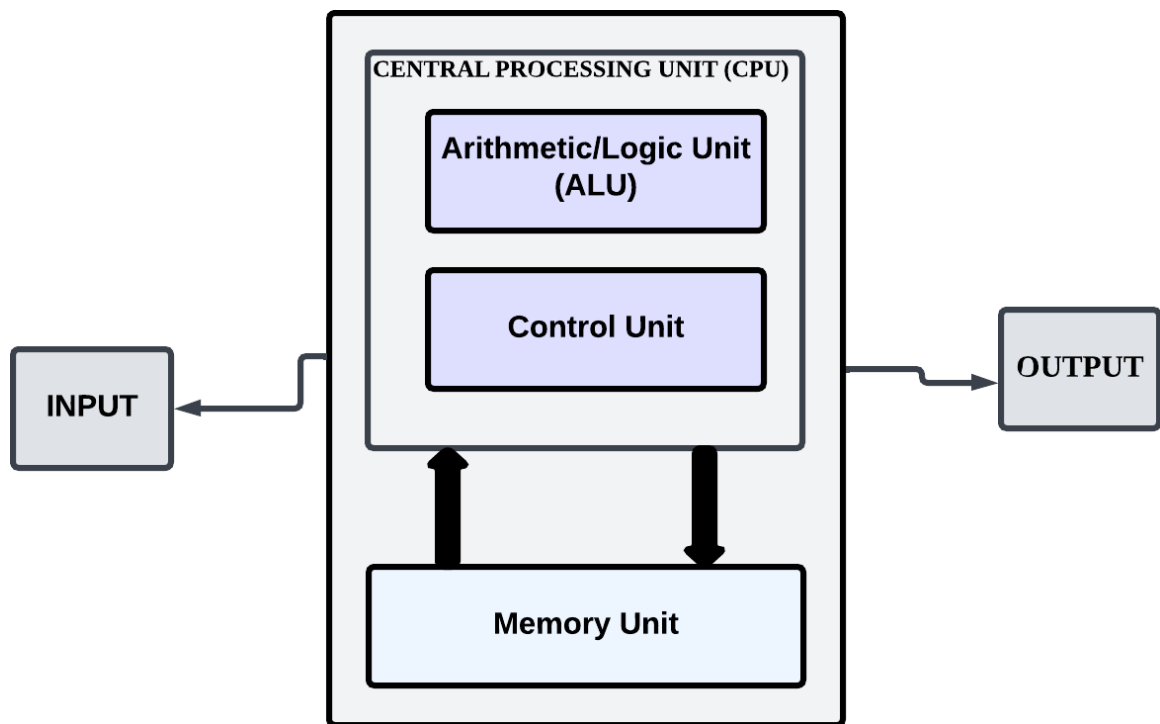
1.1 Architecture For General Purpose Processors

Classically, the Von-Neumann architecture [1] in Figure 1 was the primary architecture that has had a profound impact on the domain for ages. Figure 1 illustrates not only the overall framework of the architecture but also each integral component. In the architecture, the three primary components of a Von Neumann architecture are i) the central processing unit (CPU), the primary device for executing instructions; ii) the memory unit - a vital repository that temporarily stores data and instructions until the CPU processes them; and iii) the input and output devices - which employ communication between users or other systems through physical interfaces. The CPU further comprises an Arithmetic/Logic Unit (ALU), along with some registers and a Control Unit. This control unit consists of an instruction register and a program counter. Although this architecture is well-known in the domain, it has several drawbacks too. The major drawback is the processor-memory bottleneck [2]. This is due to the register file and dynamic random access memory (DRAM) forming the architecture's memory block. However, large amounts of data cannot be stored in this register file; thus – an issue arises with capacity limitations. The data necessitates

utilization of the DRAM for storage; however, being a generally slower memory unit - this results in performance throttling of the entire architecture. Therefore, the responsibility for slowed-down performance within the architecture lies solely with its memory block if it fails to operate at a faster rate. For this reason, this challenge is termed the “Von-Neumann Bottleneck”.

Figure 1

The Von-Neumann Architecture



1.2 Challenges Of General Purpose Processors

In 1965, Gordon Moore originally predicted the doubling of transistor density each year [3] while in 1975, the prediction was revised, which eventually became the "Moore's Law" [4]. The law states that the number of transistors in an integrated circuit will double

every two years [5]. Since transistor density increases quadratically along with the linearly growing speed, architects opted for utilizing more transistors to enhance performance. Although Moore's Law continued for a longer time compared to Dennard scaling, the era of Dennard scaling [6] concluded first. Thus, more transistors are switching now which means it signifies more power. Also, the energy budget does not rise and the single inefficient processor has been replaced with multiple efficient cores. In this case, there are no more tricks left to keep making big advancements in cost performance and energy efficiency for general-purpose architectures. Since the energy budget is limited due to electromigration, mechanical, and thermal limits of chips, we need to reduce the energy per operation if we need to seek higher performance. Given this overhead, slight modifications to present cores might give us 10 percent enhancements. However, if we desire a leap in performance by one order of magnitude while retaining programmability then we must increase the amount of arithmetic operations per instruction from just one to hundreds. The demise of Dennard scaling states that the computer architect has to find a new way to exploit instruction level parallelism (ILP), which was the primary architectural method to gain performance of the device. The catch is that the instruction level parallelism caused greater inefficiency. To keep the pipeline full, the branches are predicted and the code is speculatively placed on the pipeline of the execution. Sometimes the predictions of the branches can be beneficial or not. When it is perfect, speculation improves performance while saving energy, but when it is not, the processor can throw away the incorrect speculated instructions along with the wastage of energy and performance. This wasted energy adds up and is greater, although the processor uses additional energy to restore the previous state. And, thus the multicore era [7, 8] was born and it shifted the responsibility of finding parallelism and determining how to use it to both programmer and language system. But it also fails to resolve the energy-efficient computation that was aggravated by the end of Dennard scaling. More cores lead to more power being used almost equally as fast. Regrettably, the power getting into a processor needs to be taken out as heat. This means that multicore processors are

limited by thermal dissipation power (TDP), which is the average amount of power that a package and cooling system can remove effectively. To attain higher rates of performance improvement – similar to what we experienced in the 1980s and 1990s – it will be necessary for us to adopt fresh architectural approaches concentrating on making integrated-circuit capability utilized more efficiently. To make this kind of efficiency possible it requires a major alteration in computer architecture that moves away from general-purpose cores towards domain-specific architectures (DSAs) [9]. Domain-specific architectures (DSAs) are a possible solution to alleviate this challenge through their several properties such as customized instruction sets, pipelining and parallelism, specialized memory hierarchies, and reduced instruction overhead. DSAs are an effective tool to handle computational needs while improving performance designated for a specific task. A brief description of the domain-specific architecture has been provided in Chapter 2.

1.3 Solution Of Ordinary Differential Equations Using Numerical Methods

Often referred to as nature’s language, differential equations [10] provide a way to describe and understand many natural phenomena in a precise and systematic way. Differential equations capture the dynamics of physical systems and are fundamental to progress in the computational sciences. It is known that systems of differential equations can be solved analytically to obtain exact solutions. Yet often, we cannot achieve this due to the intricate nature of the systems under study. For workloads that involve high-performance computation, ordinary differential equations (ODEs) are often utilized, but modern general-purpose processors - referred to as CPUs - generally offer a modest throughput for solving these equations. This is where the power of scientific computing and domain-specific architectures can be leveraged. In approximating solutions to differential equations and effectively exploring their behavior, numerical methods are often instrumental.

1.4 Overview Of The Proposed Work

Firstly, this research delves into the implementation of a simple hardware accelerator for an application for obstacle detection awareness for disabled individuals. It involves working with the riscv64 compiler, a customizable and reconfigurable RISC-V-based system-on-chip (SOC) design tool, "Chipcron", and various open-source tools like Icarus Verilog simulator [11], gtkwave waveform viewer [12], and yosys synthesizer [13]. This approach leads towards designing and creating a gate-level netlist for the tape out in case of any hardware accelerators meant to perform for a specific workload. After that, the research focuses primarily on the design and development of hardware accelerators for solving ordinary differential equations using different numerical methods. The preliminary research concentrated on the implementation of a hardware accelerator design for two numerical methodologies, such as the Euler and Modified Euler methods, for solving ordinary differential equations. The work has been specifically done using Very High-Speed Integrated Circuit Hardware Descriptive Language (VHDL) in Xilinx Vivado Software and the AXI4 stream single-precision floating-point IP units from Xilinx Vivado have been used to implement and analyze the hardware accelerators specifically for these two methods. The accelerator is typically deployed on the Zynq ZC702 FPGA Evaluation Kit.

Chapter 2

Background

This chapter will discuss in-depth reviews of two specific areas: i. Domain-specific Architectures and ii. RISC-V micro-architecture.

2.1 Domain-Specific Micro-Architecture

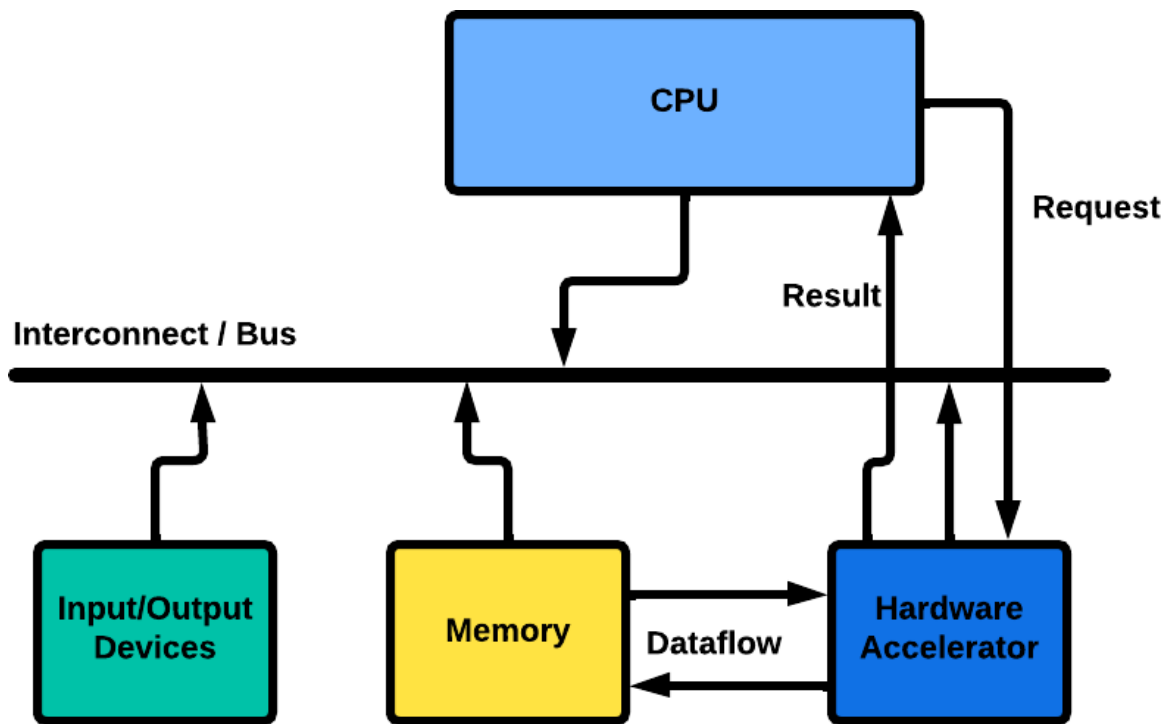
Several years ago, hardware manufacturers grappled with complexities in their architecture: power dissipation; leakage current; and wire delays - issues that caused significant setbacks [14–17]. During this period, stagnation arose in the improvement of the single-core micro-architecture, prompting a shift of attention towards new design models. These new design models give rise to the “multi-core” micro-architecture. This approach involves the incorporation of multiple cores onto a single die (functional electronic components fabricated on a thin piece of silicon) [18]. However, the approach was short-lived and prompted the manufacturers to utilize more compact and efficient designs to deal with high computing demands.

To reconcile the conflicting demands of computing power and performance, many computer architects embraced a solution, which was later termed “domain-specific architectures” (DSAs) or “hardware accelerators” [19]. The graphics processing unit (GPU) sector has notably flourished over the last decade through the successful implementation of this strategy. DSAs are the configured hardware for a specific application or workload that aims to deliver improved performance, efficiency, and low computing power by often outperforming general-purpose central processing units (CPUs) in their designated tasks. This approach helps to minimize CPU overload and memory space consumption, meeting the evolving demands of computational tasks in various domains of scientific computing [20–24]. In Figure 2, an overview of a hardware accelerator architecture has been

presented [20], demonstrating how the central processing unit (CPU) 'offloads' its task to the accelerator in this diagram. Subsequently, executing the task faster than the CPU and delivering results directly back to the CPU. DSAs might have different forms such as Application-specific integrated circuits (ASICs), Field-programmable gate arrays (FPGAs), Graphics Processing Units (GPUs), and Tensor Processing Units (TPUs).

Figure 2

Overview Of An Architecture Of A Hardware Accelerator (adopted from [16])



2.2 RISC-V Micro-Architecture

RISC-V [25] is an open-source instruction-set architecture (ISA) that follows the concept of Reduced Instruction set architectures (RISC). It holds high value because of the key properties mentioned below:

a. Instruction set architecture (ISA): A collection of basic instructions like arithmetic and logic operations or data transfer tasks that a processor can execute.

b. Base Integer ISA: RISC-V can handle base integer instructions, which include tasks like integer arithmetic and logic, data shifting or moving, controlling the flow of programs, and system calls.

c. Modular Extensions: RISC-V additionally has support for various modular extensions such as Multiply/Division (M), Floating-point (F), Vectors (V), Atomic (A), Bit Manipulation (B), and Cryptographic (C) extensions.

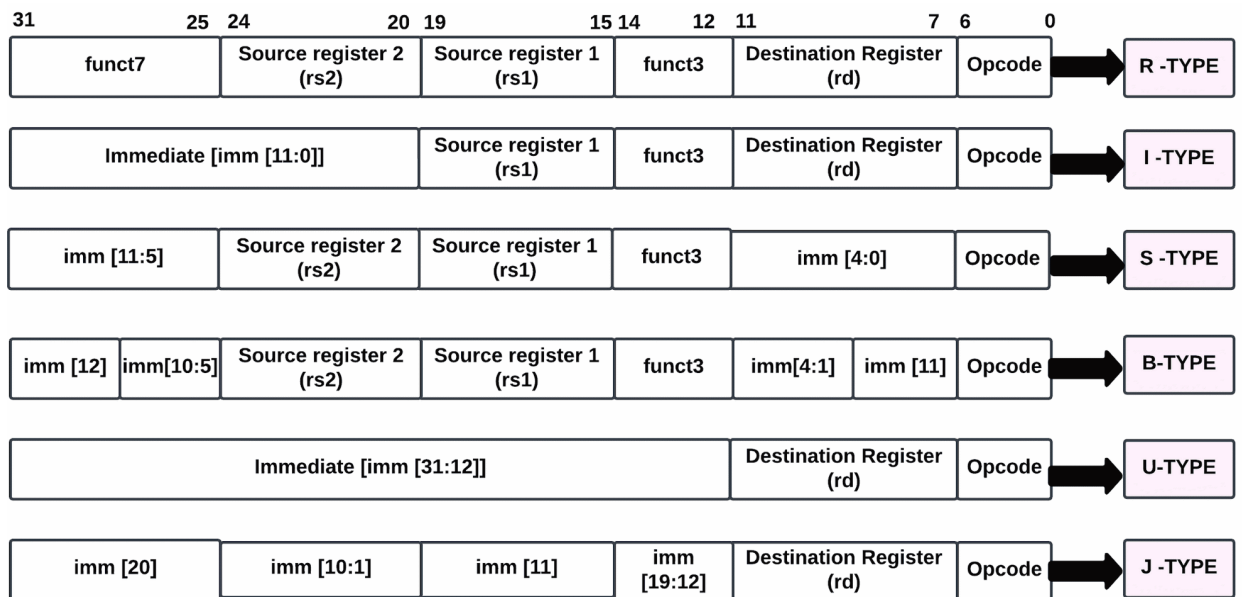
d. Register Files: There are 32 registers in RISC-V, with each register having a size of 32 bits in the case of RV32I, 64 bits for RV64I, and 128 bits for RV128I instruction set formats. These are utilized to keep temporary values and operands while doing computations.

e. Pipelined Architecture: RISC-V processors often use a pipelined architecture to enhance the speed of instruction flow. This pipeline includes different stages like fetching an instruction, decoding it, executing the operation, accessing memory, and finally writing back results. The process of pipelining lets many instructions be handled at once, making things work faster overall.

The RISC-V instructions can be classified into several formats such as R-type (Register-type), I-type(Immediate-type), S-type(Store-type), B-type(Branch-type), U-type(Upper-Immediate-type), and J- type(Jump-type) [26]. The RISC-V instruction sets are given in Figure 3.

Figure 3

RISC-V Instruction Sets



Chapter 3

Literature Review

The intersection of micro-architecture and differential equations is observed primarily in two distinct areas. Specifically, they are:

- Acceleration of scientific computing workloads based on systems of differential equations with commercially pre-built accelerators.
- Acceleration of scientific computing workloads based on systems of differential equations with customized, application-specific hardware accelerators.

An in-depth overview of the current methodologies and implementation of hardware accelerators is presented below.

3.1 Commercially Pre-Built Accelerators For Scientific Computing Workloads

Commercial pre-built accelerators such as "general-purpose graphics processing units" (GPUGPUs) have played a vital role in scientific computation because they offer parallel computing capabilities. The researchers in [27] delve into the potential of graphics processing units (GPUs) for enhancing speed during scientific calculations, notably those associated with Monte Carlo simulations of classical spin models in statistical mechanics. The research findings underscore that careful algorithm optimization for specific GPU architectures can yield significant speed-ups, often exceeding two orders of magnitude, emphasizing the importance of GPUs in the future of computational science. Another study [28] suggests that optimizing the DLR TAU code [29] by profiling the most time-consuming processes, testing new partitioning algorithms, and exploring hardware acceleration with GPUs a crucial components for performance acceleration. Simulations for computational fluid dynamics (CFDs) have been readily ported to the GPGPU paradigm.

Methods for improving an unstructured grid solver's efficiency on modern graphics

hardware are proposed in [30], with a focus on the three-dimensional Euler equations for inviscid, compressible flow. The performance of the solver based on two benchmark cases, such as a National Advisory Committee for Aeronautics wing (NACA0012) and a missile has been demonstrated. The implementation of these techniques resulted in an average speed-up factor of roughly $9.5\times$ over parallelized OpenMP code running on a quad-core CPU and about $33\times$ over the equivalent serial code, indicating potential advantages within computational fluid dynamics problem-solving tasks.

In general, software tools for fluid dynamics modeling, analysis, and simulation are largely available to the community [31]. Researchers in [32] have harnessed the inherently parallel structure of GPUs to enhance combustion modeling - a move that resulted in significant acceleration of simulations and improvements in predictive capability. The GPU-enhanced algorithms significantly outperform CPU-only simulations, with computational time scaling less favorably on CPUs, and GPUs are expected to play an increasingly important role in future combustion modeling. Although not explored at length, there are some prior works focused on porting generalized multigrid solvers to GPUs [33].

3.2 Customized, Application-Specific Hardware Accelerators For Scientific Computing Workloads

The work done in this area cuts across multiple problem domains. The research in [34] employs an FPGA-based system to simulate two-dimensional Ising lattices, achieving considerable speed improvements over single CPU, single GPU, and prior FPGA systems. A superior hybrid random number generator has been introduced in this study and demonstrates the value of FPGAs for scientific computing, particularly Monte Carlo simulations. Another study in [35] introduces a methodology for optimizing FPGA implementations of digital signal processing (DSP) circuits to minimize hardware area while maintaining a minimum throughput constraint. By combining module selection and resource-sharing techniques during pipeline synthesis, two different exploration algorithms yield

more efficient results compared to when either technique is used independently.

A novel framework for the real-time evaluation of one-dimensional computational fluid dynamics (1D-CFD) models is presented in [36], deployed on a field-programmable gate array (FPGA) to enhance fuel pressure estimation in diesel engines. Demonstrating significant resource savings and scalability, the approach uses precision-timed (PRET) processor cores and a configurable, heterogeneous architecture to meet real-time constraints and model a diesel fuel system efficiently. FPGA-based computing systems have also been designed to accelerate iterative linear-equation solvers [37], achieving high performance with low memory bandwidth. The implemented prototype, using a pipelining approach across multiple FPGAs, demonstrated linear scalability and outperformed software execution on a microprocessor.

The Johnson and Moon equation and the Mackey-Glass equation, both nonlinear time-delay systems, have been implemented on an FPGA [38]. Using the fourth-order Runge-Kutta method and 32-bit IEEE-754-1985 floating-point standards, the systems are implemented on a Xilinx Virtex-6 FPGA, with maximum frequencies of around 432 MHz and results that align with numerical simulations. Alongside the implementation, the numerical algorithms themselves are also major contributors to performance [39]. The research findings from FPGA-based deployment in [28] show significant computational gains from optimization, better scalability with the graph partitioner algorithm, and promising results from a mixed hardware-software computation platform for simulations.

Computational fluid mechanics problems are a prime candidate for customized coprocessor-based acceleration, especially in the aerospace domain. The authors of [40] introduce a novel hardware solver based on the open-source RISC-V instruction set architecture for solving ordinary differential equations (ODEs). The study designs reconfigurable coprocessors that implement Euler and Runge-Kutta numerical methods for ODE solving. The coprocessor achieves a remarkable 4.8x performance improvement in the best-case scenario, with only a marginal increase of 13.3 percent in hardware resources and 8.1 per-

cent in additional power dissipation. A high-performance hardware accelerator is designed to improve the solution speed for ordinary differential equations (ODEs) in [41] for use in high-performance computing applications. The implemented hardware accelerator on an FPGA board demonstrates up to 14x speedup compared to a single-core CPU solution, although speedup decreases with increasing ODE-solver complexity or precision due to reduced parallelism on the FPGA. The topic of generalized multigrid solvers accelerated by hardware has been explored to a minimal extent. The researchers present a hardware implementation of the V-cycle Multigrid method for solving the 2D-Poisson equation using FPGAs, demonstrating greater performance compared to similar iterative solvers and a C++ version of the algorithm. The Multigrid hardware solution notably outperforms its software counterpart, offering significant speedups, particularly for smaller problem sizes [42].

Chapter 4

Motivation

Given the shortcomings of general-purpose processors in making performance better, be it through ILP techniques or methods using multicore, and also with Dennard scaling and Moore's Law no longer effective - it seems quite unlikely that processor architects and designers will succeed in maintaining significant rates of performance enhancements for general-purpose processors. A clear choice that can be seen as long-lasting is "domain-specific computing" or what some might call "hardware acceleration". Hardware acceleration emerges as a crucial tool for a wide range of computational tasks across various domains due to the increased benefits in energy efficiency, parallel processing capabilities, and better performance (high accuracy with less processing time). The term "parallel processing capabilities" refers to the system's ability to perform multiple tasks simultaneously. The parallel processing competencies of those accelerators allow overall performance improvements in workloads that contain large and complicated numerical computations [43]. Parallel processing also involves decomposing a larger computationally complex task into smaller concurrent problems with careful consideration of data sharing, communication, and dependencies between parallel tasks to ensure effective and accurate results. The research in [44] pointed out, that memory access has turned into a costly operation when compared with arithmetic computations. Accessing a block within a 32-32-kilobyte cache takes approximately 200× more energy than adding up a 32-bit integer, then it is quite evident that we must optimize these accesses to lower energy consumption. DSAs can utilize memory hierarchy more efficiently. DSAs have an additional characteristic that they can work with less precision when it is enough. Regular CPUs usually possess 32 and 64-bit integers along with floating-point (FP) data. In numerous cases of machine learning and graphics tasks, this accuracy could be excessive. For instance, in deep neural networks

(DNNs), when we do inference, we often use 4-bit, 8-bit, or sometimes 16-bit integers that help increase the throughput of both data and computation [45]. For applications related to DNN training also it assists but is not necessary; using 32 bits works well while at times only employing 16 bits is enough. Finally, another critical factor necessary for high-performance scientific simulations is the system's scalability. Systems with parallel processing capabilities can branch out to satisfy growing computing needs or scale up to tackle bigger issue sizes.

Exploring numerical method solutions based on hardware acceleration and understanding the associated challenges in their development remains a crucial task [46,47]. This approach helps in assessing the current state of hardware-accelerated scientific computing, identifying future directions, and potentially uncovering breakthroughs within this rapidly evolving field. In addition, this study also seeks to (1) highlight the synergies between different disciplines like pure science, theoretical mathematical concepts (especially those of the realm of differential equations), computer science, and engineering, and (2) scrutinize how their integration can lead to ground-breaking improvements in computational science. This holistic perspective is essential for fostering innovation and enhancing the capability to tackle complex scientific problems with greater precision and efficiency. This research also delves into the investigation based on the intricate relationship between differential equations and hardware acceleration in the realm of scientific computations.

Chapter 5

Methods And Implementation

This chapter delves into a small hardware accelerator design that can be controlled from a C program and also with RISC-V32I ISA format using a reconfigurable and customizable RISC-V System on Chips (SOC) tool, “Chipcron Tool”. Now, the question arises how does our C program turn into a hardware representation? In Figure 4, the flow diagram from application software to hardware implementation has been represented. Whenever we are writing any C program, that program typically serves as our “Source Code”. The output of the source code will be translated into assembly language (or instructions) using any compiler. The compiler will turn our code into specific instructions through a “.exe” file. The assembler then takes our .exe file and converts it into binary, which is termed the “machine language” or “machine code” (represented using logic ‘0’s and ‘1’s). The machine code will then be fed to the hardware and accordingly, it generates the output and will be used for tape out.

In [48], the design of the hardware accelerator specific for obstacle detection for disabled individuals has been implemented. The whole processor for designing the accelerator has been described. At first, the C code for our accelerator will be written with some inline assembly instructions. The snippet for the C code is provided below in Figure 5. In this code, we have initialized some variables such as ‘sensor’, ‘buzzer’, “buzzer_reg”, and “mask”. These variables are used to store sensor readings and buzzer states respectively. The variable sensor is taken as an input and the buzzer is served as an output in the code. The ‘mask’ is initialized with some value, 0xFFFFFFFF, generally for some bitwise operations. The assembly inline instructions, “and”, and “or” have been used to perform bitwise operations on the register “x30” using the values of “buzzer_reg” and mask. Depending on the value of the sensor, the code will set or reset the buzzer. Finally, the code will update

the register “x30” with the new values of “buzzer_reg”. Then, the RISC-V compiler and SPIKE simulator have been used to compile the code. The Spike simulator, specifically designed to execute RISC-V assembly programs for various RISC-V versions, has been used. It is an open-source RISC-V ISA (Instruction Set Architecture) simulator developed primarily by the RISC-V project. The following command will be used to compile the code using RISC-V Compiler:

```
→ riscv64-unknown-elf-gcc -march=rv64i -mabi=lp64 -ffreestanding -o output assembly.c
```

The simulation result when the sensor value is 0 and 1 has been given in Figure 6 and Figure 7. In Figure 6, when the sensor value is 0, i.e., if there’s no object, the buzzer will throw as 0. Similarly, in Figure 7, the buzzer will be 1 if any object is detected through the sensor. All the results will be generated using the SPIKE simulator through the following command:

```
→ spike pk output
```

The C code will be converted into assembly instructions using the two commands specific to the RISC-V compiler: → riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32 -ffreestanding -nostdlib -o out assembly.c

```
→ riscv64-unknown-elf-objdump -d -r out ; obstacledetection.txt
```

The Disassembly of the machine instructions into their assembly instructions has been represented in Figure 8. In the first column, the address of the ‘main’ program has been generated through the RISC-V compiler. In the figure, it has been seen that the address of the main program or the program counter(PC) has been started from “00010054” and it is increasing by 4, as the program counter (PC) = PC + 4.

The next column shows the machine instructions used for the particular assembly instructions. It symbolizes the instruction in machine code, given as a hexadecimal value. Normally, in machine language, commands are coded into binary numbers that the processor can immediately execute. Hexadecimal representation is frequently employed to show

machine code because it gives a smaller, easier-to-read way of presenting these binary instructions. In the third and fourth columns, the RISC-V assembly instructions have been shown according to the C code of our application.

The assembly instructions will be then fetched into a javascript object notation (.json) file with the specific instructions used for our assembly program. A snippet of the .json file has been given in Figure 9. The .json file consists of several RISC-V instructions based on the RISC-V architecture and it is specifically designed for a 3-stage pipelined processor, which will be generated by the Chipcron tool in the next step.

The Chipcron tool will generate the processor and its corresponding testbench according to our specific needs of the application. A portion of the Verilog file of the processor generated by the Chipcron tool is given below in Figure 10. Icarus Verilog is a simulation and synthesis tool for Verilog that comes as open-source. Verilog is a hardware description language (HDL), used to design and simulate digital circuits. With Icarus Verilog or iverilog, users can write code in the form of Verilog which describes digital circuits and then give it to simulate how these circuits will behave. In the case of GTKWave, it is an open-source waveform viewer, which can be freely accessed and used. It assists in analyzing and visualizing the results of simulations from digital circuit simulations. Users frequently utilize it with Verilog and VHDL simulators like Icarus Verilog to watch waveforms that are produced during simulation runs. After the generation of our processor and test bench, the functional simulation using open-source simulators like Icarus Verilog and gtkwave will be generated, which is given in Figure 11. Functional simulation is a process to identify if the behavior of our given program is generating the result correctly or not. According to the figure, it has been seen that what we are expecting, is giving the result as “PASS”, which states our design is functioning correctly according to our requirements.

The next step is the synthesis of our design. In simple terms, synthesis is converting a system’s description that is at a high level into its representation at a lower level. This implementation can then be used in hardware. The design of the hardware accelerator will

then be synthesized using Yosys synthesizing tool using the SKY130 Process design kit (PDK), an open-source PDK developed for the SkyWater 130 nm semiconductor process in Figure 12. After utilizing and adjusting the static ram (SRAM) from the PDK to our design, the gate-level synthesized netlist will be generated in Figure 13. These synthesized net-lists state that the design is working correctly after synthesis. These netlists will be utilized for the tape out of our specific application or task after physical design procedures [48].

Figure 4

The Flow Diagram From Application Software To Hardware Representation

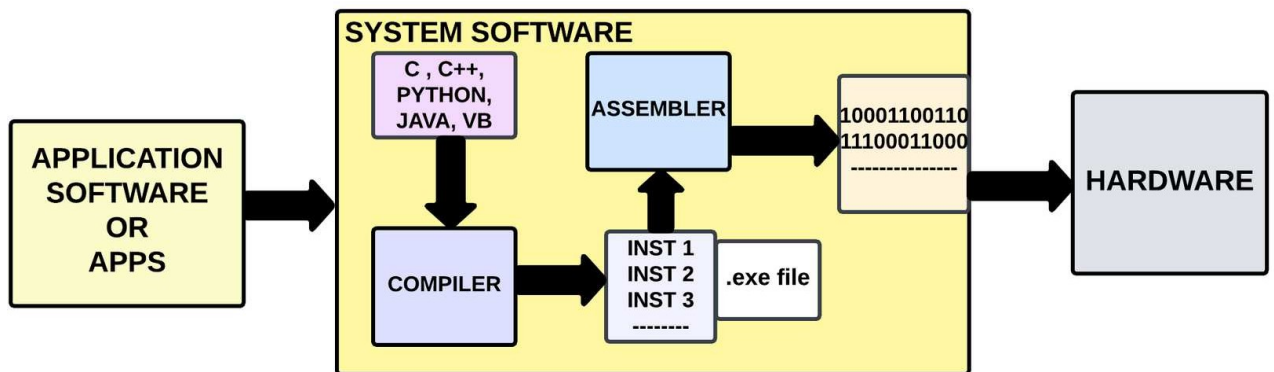


Figure 5

A Snippet Of The C code For The Obstacle Detection With Inline Assembly Instructions

```
 4  int main () {
 5      int sensor ;
 6      int buzzer=0 ;
 7
 8      int buzzer_reg ;
 9      int mask = 0xFFFFFFFF;
10      buzzer_reg = buzzer * 2;
11
12      asm volatile(
13          "and x30, x30, %1\n\t"
14          "or x30, x30, %0\n\t"
15          :
16          : "r" (buzzer_reg), "r"(mask)
17          : "x30"
18          );
19
20      while(1) {
21
22          asm volatile(
23              "andi %0, x30, 0x01\n\t"
24              : "=r" (sensor)
25              :
26              :);
27
28          if (sensor) {
29              buzzer = 1;
30              buzzer_reg = buzzer * 2;
```


Figure 8

Disassembly Of The Machine Instructions Into Their Assembly Instructions

```
1
2  out:    file format elf32-littleriscv
3
4
5  Disassembly of section .text:
6
7  00010054 <main>:
8      10054:    fe010113      addi    sp, sp, -32
9      10058:    00812e23      sw     s0, 28(sp)
10     1005c:    02010413      addi    s0, sp, 32
11     10060:    fe042623      sw     zero, -20(s0)
12     10064:    ffd00793      li     a5, -3
13     10068:    fef42423      sw     a5, -24(s0)
14     1006c:    fec42783      lw     a5, -20(s0)
15     10070:    00179793      slli   a5, a5, 0x1
16     10074:    fef42223      sw     a5, -28(s0)
17     10078:    fe442783      lw     a5, -28(s0)
18     1007c:    fe842703      lw     a4, -24(s0)
19     10080:    00ef7f33      and    t5, t5, a4
20     10084:    00ff6f33      or     t5, t5, a5
21     10088:    001f7793      andi   a5, t5, 1
22     1008c:    fef42023      sw     a5, -32(s0)
23     10090:    fe042783      lw     a5, -32(s0)
24     10094:    02078663      beqz   a5, 100c0 <main+0x6c>
25     10098:    00100793      li     a5, 1
26     1009c:    fef42623      sw     a5, -20(s0)
27     100a0:    fec42783      lw     a5, -20(s0)
28     100a4:    00179793      slli   a5, a5, 0x1
```

Figure 9

A Portion Of The Conversion Of The Assembly Instructions To Json File Format

```
1      {
2          "ALU_dist": 3,
3          "pc_bit_width":8,
4          "value_bit_width":32,
5          "data_mem_bit_width":8,
6          "immediate":12,
7          "address_size":5,
8          "shamt":5,
9          "instructions":{
10             "LUI"    :false,
11             "AUIPC"  :false,
12             "JAL"    :true,
13             "JALR"   :false,
14             "BEQ"    :true,
15             "BNE"    :false,
16             "BLT"    :false,
17             "BGE"    :false,
18             "BLTU"   :false,
```

Figure 10

Generation Of The Verilog Files Of The Processor Design And Its Associated Testbench Using The Chipcron Tool

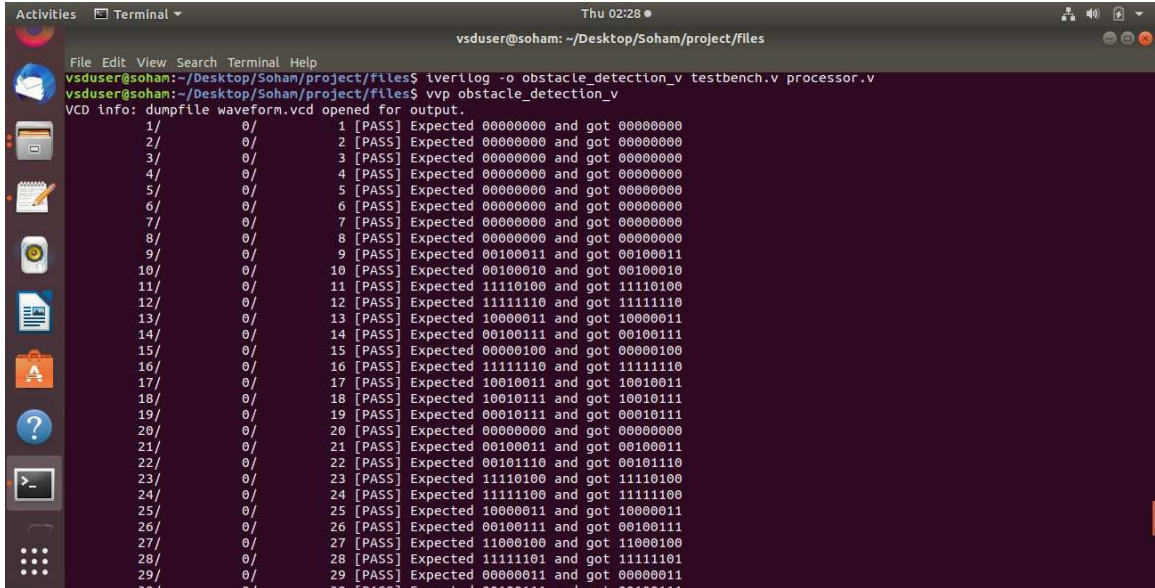
```
endmodule

module wrapper(clk, resetn, uart_rxd, uart_rx_en, uart_rx_break, uart_rx_valid, uart_rx_data, output_gpio_pins, input_gpio_pins, w
input clk;
output reg write_done ;
output reg [2:0] instructions ;
input wire input_gpio_pins;
output reg output_gpio_pins;
reg rst;
reg neg_clk;
reg neg_rst;
input resetn ; // Asynchronous active low reset.
input uart_rxd ; // UART Recieve pin.
input uart_rx_en ; // Recieve enable
output uart_rx_break; // Did we get a BREAK message?
output uart_rx_valid; // Valid data recieved and available.
output [7:0] uart_rx_data ; // The recieved data.

wire web;
wire [7:0]inst_mem_addr;
wire [7:0] data_mem_addr;
wire [31:0]data_mem_wdata;
wire [3:0]mem_wstrb;
wire [31:0] inst_mem_rdata;
wire [31:0] inst_mem_rdata_dummy;
reg [31:0] inst_mem_rdata_reg;
```

Figure 11

Simulation Using VVP Tool Through Icarus Verilog



```
vsduser@soham: ~/Desktop/Soham/project/files
File Edit View Search Terminal Help
vsduser@soham:~/Desktop/Soham/project/files$ iverilog -o obstacle_detection_v_testbench.v processor.v
vsduser@soham:~/Desktop/Soham/project/files$ vvp obstacle_detection_v
VCD info: dumpfile waveform.vcd opened for output.
 1/ 0/ 1 [PASS] Expected 00000000 and got 00000000
 2/ 0/ 2 [PASS] Expected 00000000 and got 00000000
 3/ 0/ 3 [PASS] Expected 00000000 and got 00000000
 4/ 0/ 4 [PASS] Expected 00000000 and got 00000000
 5/ 0/ 5 [PASS] Expected 00000000 and got 00000000
 6/ 0/ 6 [PASS] Expected 00000000 and got 00000000
 7/ 0/ 7 [PASS] Expected 00000000 and got 00000000
 8/ 0/ 8 [PASS] Expected 00000000 and got 00000000
 9/ 0/ 9 [PASS] Expected 00100011 and got 00100011
10/ 0/10 [PASS] Expected 00100010 and got 00100010
11/ 0/11 [PASS] Expected 11110100 and got 11110100
12/ 0/12 [PASS] Expected 11111110 and got 11111110
13/ 0/13 [PASS] Expected 10000011 and got 10000011
14/ 0/14 [PASS] Expected 00100111 and got 00100111
15/ 0/15 [PASS] Expected 00000100 and got 00000100
16/ 0/16 [PASS] Expected 11111110 and got 11111110
17/ 0/17 [PASS] Expected 10010011 and got 10010011
18/ 0/18 [PASS] Expected 10010111 and got 10010111
19/ 0/19 [PASS] Expected 00010111 and got 00010111
20/ 0/20 [PASS] Expected 00000000 and got 00000000
21/ 0/21 [PASS] Expected 00100011 and got 00100011
22/ 0/22 [PASS] Expected 00101110 and got 00101110
23/ 0/23 [PASS] Expected 11110100 and got 11110100
24/ 0/24 [PASS] Expected 11111100 and got 11111100
25/ 0/25 [PASS] Expected 10000011 and got 10000011
26/ 0/26 [PASS] Expected 00100111 and got 00100111
27/ 0/27 [PASS] Expected 11000100 and got 11000100
28/ 0/28 [PASS] Expected 11111101 and got 11111101
29/ 0/29 [PASS] Expected 00000011 and got 00000011
```

Figure 12

The Internal Representation Of The Accelerator Design In The Form Of A Netlist Using Yosys Synthesizing Tool

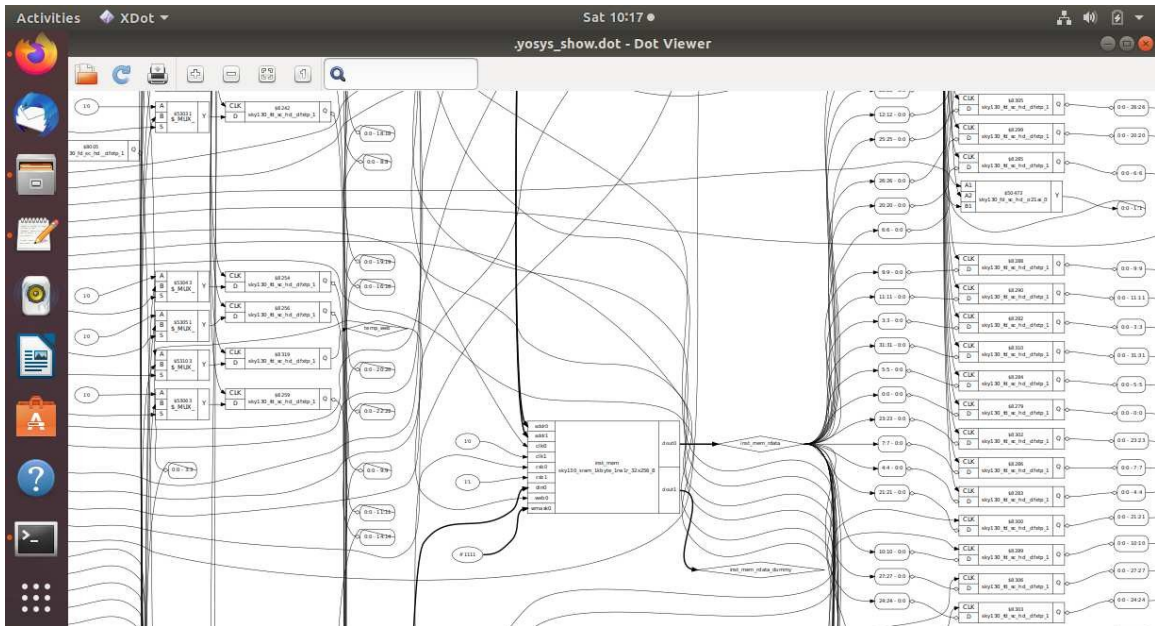
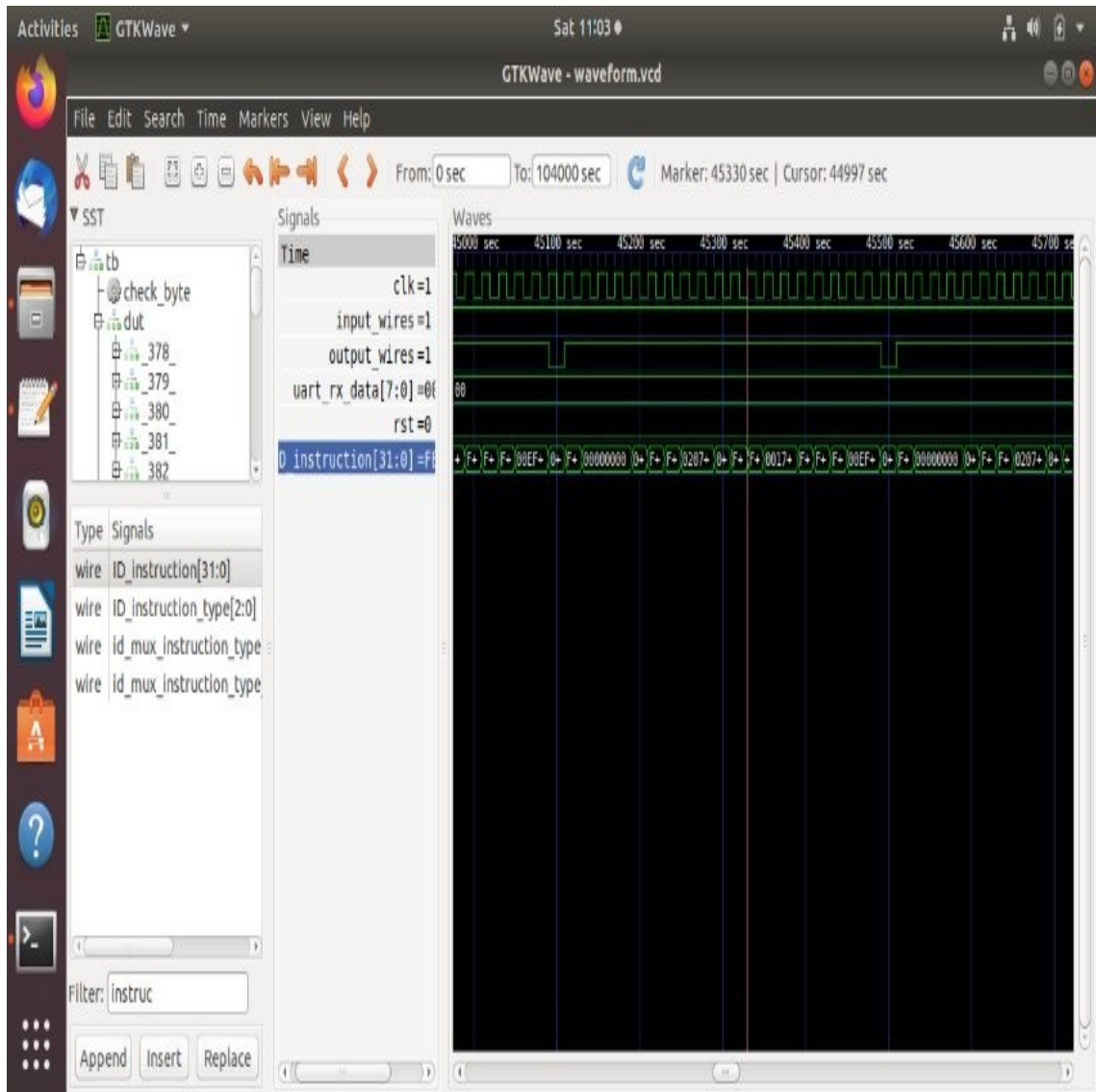


Figure 13

Gate-level Synthesis Simulation Results Of The Hardware Accelerator



Chapter 6

Numerical Methods For Solving Ordinary Differential Equations

Several numerical methods exist to solve ordinary differential equations. Our preliminary research focused on two simple yet efficient methods for solving ordinary differential equations: the Euler method and its counterpart, the Modified Euler method. An overview of these two numerical approaches has been explored below.

6.1 The Euler Method

The Euler method was first proposed in 1768 by Leonardo Euler [49]. This methodology is a first-order numerical method generally used to solve ordinary differential equations with a pre-defined initial value. Here, the term “first-order” refers to the local error or more specifically the error per step, which is directly proportional to the square of the step size. Furthermore, the global error, or the error at a given time, is directly proportional to the step size. Constructing more complex numerical methods can employ this applicable method. Now, the question that naturally surfaces: what makes it so useful? Primarily, it is simple, and with the ease of its implementation and understanding, it is computationally efficient for solving a wide range of ODEs, making it an ideal choice for numerous modeling and simulation tasks. The structural analysis field, computational biology sector, chemistry domain, and computer graphics arena all frequently utilize the Euler method; its applications extend even further. The Euler method can be represented as:

$$y_{n+1} = y_n + hf(x_n, y_n) \quad (1)$$

$$x_{n+1} = x_n + h \quad (2)$$

In equation (1), $f(x_n, y_n)$ represents the function for the ODE. 'h' is the step size. The Euler method is generally used to approximate the value of 'y' with a given initial value of 'x'.

6.2 The Modified Euler Method

The modified Euler method, also known as the 'Heun method', is another numerical method for solving ODEs [50]. It offers more accuracy in approximations when compared with the Euler method. Generally, this approach considers not only the initial estimate based on the local slope but also an average slope over the step. Computational chemistry, physics, biology, etc are some of the applications of the modified Euler method. The following equations represent the Modified Euler method.

$$y_{n+1} = y_n + hf(x_n+h/2, y_n+h/2*f(x_n, y_n)) \quad (3)$$

$$x_{n+1} = x_n + h \quad (4)$$

Here, $n = 0, 1, 2, \dots, (x - x_0)/h$, h is the step size, $f(x_n, y_n)$ denotes the function in the equation, specified in (3).

6.3 Runge-Kutta Methods

Runge-Kutta numerical methods are widely used to solve ordinary differential equations (ODEs). Among the Runge-Kutta methods, second-order, third-order, and fourth-order Runge-Kutta solvers are commonly used variants for approximating the solutions to the differential equations. The second-order Runge-Kutta numerical method or the "RK2" method [51] is a simple, yet effective method, that updates the solution of each step by approximating the derivative at a particular location in the ODE with a weighted average of two slopes. This approach is thereby garnering favor due to its reasonable accuracy and

high computing efficiency in numerous real-world applications. In comparison to RK2, the third-order Runge-Kutta technique, or the “RK3” method, [52] offers better accuracy. In this technique, to approximate and update the result, three slopes and their weighted averages need to be calculated. As a result of the increased analyzing effort, the derivative is approximated more accurately, which makes it appropriate for ODEs with steeper gradients or when greater accuracy is needed. Runge-Kutta fourth order differential equation solver method, also known as the “RK4” method [53] is used to approximate the value of ‘y’ with a given value of ‘x’. RK4 methods are typically used to resolve the initial-value problems of differential equations [54]. The general form to describe the fourth-order Runge-Kutta method to solve an ODE is given below:

$$K_1 = f(x_n, y_n) \quad (5)$$

$$K_2 = f(x_n + h/2, y_n + hK_1/2) \quad (6)$$

$$K_3 = f(x_n + h/2, y_n + hK_2/2) \quad (7)$$

$$K_4 = f(x_n + h, y_n + hK_3) \quad (8)$$

and,

$$y_{n+1} = y + (h/6)(K_1 + 2K_2 + 2K_3 + K_4) + O(h^5) \quad (9)$$

$$x_{n+1} = x_n + h \quad (10)$$

The accelerator will be made up of several modules, including the functional computation block (FCB), where the function of the ODE is indicated, the K-Blocks, which execute ordinary differential equations of coefficients using the FCB, and the Runge-Kutta Modules, which compute the values of x_{n+1} and y_{n+1} . Additionally, a memory unit that holds the

value of the approximation of x_{n+1} and y_{n+1} will be introduced. A control unit, to implement the custom instructions per the RISC-V ISA standard, will be implemented. The implementation includes the register file to initialize unit values into specific registers. The hardware accelerator sets some initial set of values of x and y and a third-party vendor AXI4 stream single-precision floating-point IP support from Xilinx Vivado will be utilized to generate the results for all the respective values for the four coefficients K_1 , K_2 , K_3 , K_4 . After issuing the necessary single custom instructions to initialize, flush, update, and store for each case following the RISC-V ISA format; the accelerators will execute 'n' iterations. In numerical methods, floating-point representation is a crucial concept that exhibits an impressive capability to represent a wide range of real numbers. The number of significant digits or bits that may be utilized to represent the fractional component of a floating-point integer is referred to as precision in floating-point arithmetic. It establishes how accurately real numbers may be represented in the limited memory of a computer. Small inaccuracies can add up and compromise the accuracy of findings in numerical computations, hence the precision of a floating-point representation is essential. The standard defines three basic formats for floating-point representation: a. Half-precision; b. Single- Precision and c. Double-Precision. Balancing computing efficiency and accuracy determines an appropriate level of precision in numerical techniques. Though it uses more memory and processing power, higher precision typically yields more precise results. Furthermore, it is crucial to comprehend the limitations of floating-point accuracy for minimizing issues like rounding errors and numerical instability in numerical computations. This work also conducts a design-space exploration [55] for the half, single, and double-precision floating-point architecture of the hardware accelerator. Additionally, it will also evaluate the performance in terms of power consumption, hardware resource usage, and timing analysis for all three floating-point precision formats in Runge-Kutta numerical solvers. The main contribution of this study aims to identify which hardware implementation of the Runge-Kutta solver yields superior performance metrics at a given clock frequency.

Chapter 7

Micro-Architecture Description

7.1 Overview

The hardware accelerator shown in Figure 14 comprises a register file, a 64-bit memory unit, a control unit, and a module that will perform the solution to the differential equations using Euler or modified Euler methodologies. In our case, the ordinary differential equation for which the hardware accelerator has been implemented using the Euler method and the modified Euler method has been specified in equation (11).

$$f(x,y) = xy(xy)/2 \quad (11)$$

The register file, control unit, and memory unit have been designed identically for both the Euler and modified Euler methods, except the functional module which will perform the specific task for solving the differential equation (11) using Euler or modified Euler method. The accelerator initiates with x_0 and y_0 set at 2 and 1. The step size h has been set up to 0.1. The control unit operates the accelerator based on custom instructions, overseeing all other modules including the memory unit, register file, and Euler/modified Euler module. It employs third-party Xilinx single-precision floating-point IP support in developing this accelerator for differential equation solutions using those methods. Through x_{n+1} and y_{n+1} , the next iteration values will be generated.

7.2 Micro-Architecture Of The Accelerator

In Figure 14, for implementing the accelerator, a set of internal buses has been utilized. In the case of the register file, three inputs such as clock, source register(rs1), and

w_en (write enable) have been employed. Another input, the clock will be set as the system's clock. In our context, we focus on a specific aspect - Source Register (rs1); this is a five-bit register address utilized within the control unit's 32-bit instruction format. For developing the instruction format for the accelerator to run the operation for solving the differential equation (11), two custom instructions for each method have been implemented. The Euler method demonstrates an initialization instruction format, "Euler Initialization" (EI), which will load all the contents of x_n , y_n , h from the register file to the 64-bit memory unit. Another instruction format, "Euler Update" (EU), which only updates the contents, has been demonstrated.

The Euler Update comprises two parts: a. Flush Mode; and b. In Store Mode. When the one-bit write enable is '1', the accelerator will operate for EI and EU Store mode, whereas when it is '0', it will operate for the EU Flush mode instruction format. The Modified Euler Method follows an identical methodology. Two instruction formats have been initialized for the modified Euler method too, which are "Modified Euler Initialization" (MEI) and "Modified Euler Update" (MEU) instruction formats. The content of the register file will produce 32-bit data to x_n , y_n , h in the memory unit to generate the result for the Euler and modified Euler method. Additionally, the other two control signals such as flush, and initial (init) generated from the control unit will operate within the memory. The 'init' serves as a one-bit control signal, set to '1' for EU store mode; otherwise, it functions as '0'. In case of a one-bit flush control signal, it will flush out the contents of the x_n , y_n , and h to the Euler or modified Euler module. A 12-bit address, "addr" is used as the starting address in the memory unit to store the variables as well as Euler or modified Euler parameters. For the Euler or modified Euler method, the content of the x_n , y_n , h will be generated from the memory unit and it will generate the result x_{n+1} and y_{n+1} for each iteration and will again be stored in the memory unit in their specific addresses. After each operation, this method updates a current memory address known as "cma_out", thus explaining its internal functionality in the below section.

Table 1 provides a comprehensive description of the internal buses within the micro-architecture of the hardware accelerator of the ordinary differential equation, stated in the equation (11), employing both methods. It includes details such as bus names, sources, and destinations for each bus along with their respective functions.

Figure 14

The Micro-Architecture Design Of The Hardware Accelerator For Solving An Ordinary Differential Equation Using The Euler And The Modified Euler Methods

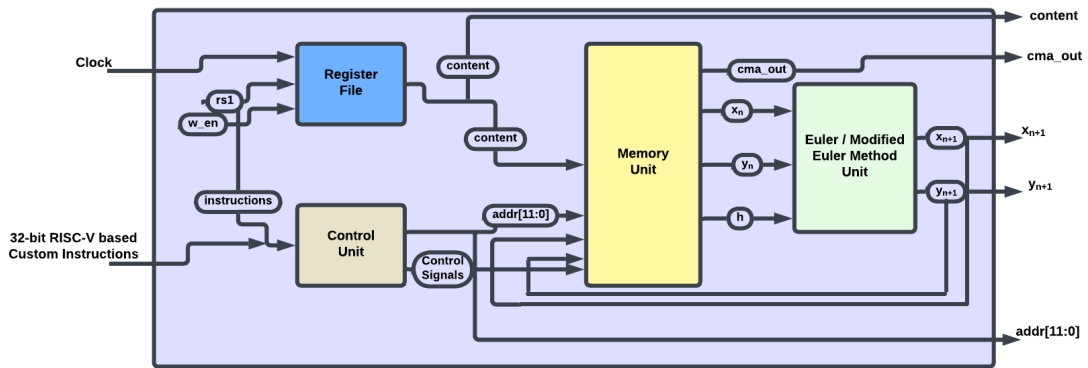


Table 1

The Comprehensive Functional Details Of The Internal Buses For Developing The Hardware Accelerator Using Both Numerical Methods

| Bus Name | Source | Destination | Bus Function |
|------------------|---------------|--------------------|---|
| write_en | Control Unit | Register File | When Write enable is 1, it will operate for EI and EU or MEI and MEU Store mode instruction formats, whereas '0' for EU and MEU Flush Mode Instruction format. |
| clock | System | Register File | Operates like an input clock. |
| Source Reg (rs1) | Register File | Control Unit | A 5-bit source register address will be used in the 32-bit instruction format of the control unit. |
| content | Register File | Memory Unit | The register file will generate 32-bit data as "contents" of x_n , y_n , h to the memory unit to generate the result for the Euler and modified Euler method modules. This will be sent to the output port "content". |

| Bus Name | Source | Destination | Bus Function |
|-----------------|---------------|--------------------------------|---|
| addr | Control Unit | Memory Unit | Starting address in the memory module to store the variables as well as the Euler and the modified Euler solvers parameters. |
| init | Control Unit | Memory Unit | A control signal of one bit. It is to be set as '1' for the EU and MEU Store Mode operation, otherwise set as '0'. |
| flush | Control Unit | Memory Unit | One-bit control signal, when it's '1', will flush out the contents of x_n , y_n , h to the Euler and the modified Euler methods module. |
| cma_out | Memory Unit | Output Port of the accelerator | Current memory address and it will update the memory address by 8 every single iteration. |

| Bus Name | Source | Destination | Bus Function |
|-----------------|---------------|---------------------------------------|-------------------------|
| x_n | Memory Unit | Euler or Modified Euler method Module | First input ' x_0 '. |
| y_n | Memory Unit | Euler or Modified Euler method Module | Second input ' y_0 '. |
| h | Memory Unit | Euler or Modified Euler method Module | Step size. |
| p_in | Memory Unit | Euler or Modified Euler method Module | $h/2$. |

| Bus Name | Source | Destination | Bus Function |
|---|---------------------------------------|--------------------|--|
| x_{n+1} | Euler or Modified Euler method Module | Memory Unit | Output ' x_{n+1} ' will generate the result for each iteration of 'x'. |
| y_{n+1} | Euler or Modified Euler method Module | Memory Unit | Output ' y_{n+1} ' will generate the result for each iteration of 'y'. |
| 32-bit RISC-V-based custom instructions | Top-level accelerator module | Control Unit | Two custom instructions will be provided to generate the outputs. |

7.3 Hardware Implementation Of The Euler And The Modified Euler Methods Modules Within The Accelerator

Both the Euler and modified Euler methods for solving the ordinary differential equation (11) share an identical functional computation block (FCB). In Figure 15, the architecture of the FCB is shown featuring support for single-precision floating-point operations. In FCB, four single-precision floating-point IPs by Xilinx Vivado have been used. We employ initial values of x and y as 2 and 1 respectively, representing x_0 and y_0 for the Euler and modified Euler methods. FPU_SUB will operate the subtraction operation in (11) while the FPU_MUL handles operations for 'xy' part in (11). Both the floating-point operations undergo another multiplication operation by employing an additional FPU_MUL floating-point block. Finally, the result 'f' will be obtained by using 'P_in' as an input for the FPU_DIV or division operator.

7.3.1 The Euler Method Module

The functional computation block (FCB) in Figure 16 generates the function and connects it to a multiply-accumulate (MAC) unit. For generating the value of x_{n+1} , a floating-point addition unit has been taken in which the initial value of 'x' will be added to the step size 'h'. The MAC unit involves the multiplication of the first two inputs and then the addition to a third input. In our case, the functional output from the FCB will be connected to the MAC unit by taking 'y' as the second input and 'h' as the third input to generate the result of y_{n+1} . This Euler module is used to generate the result of the iterations for both x_{n+1} and y_{n+1} .

Figure 15

The Functional Computation Block (FCB)

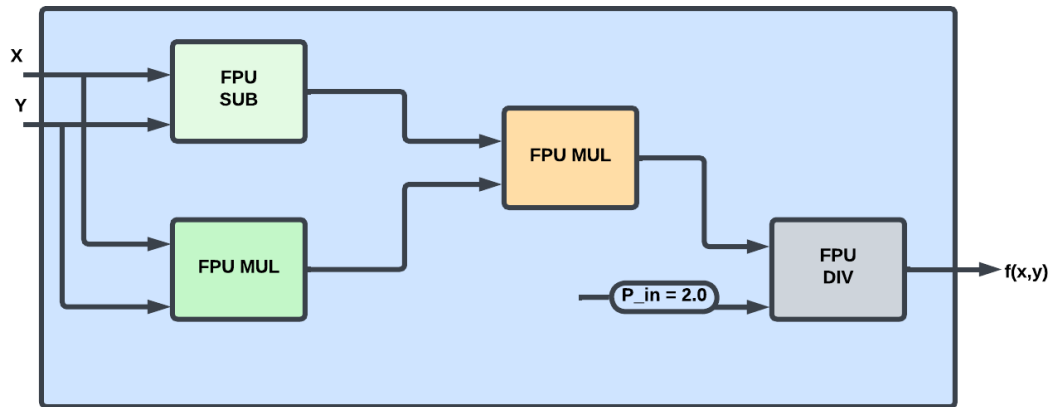
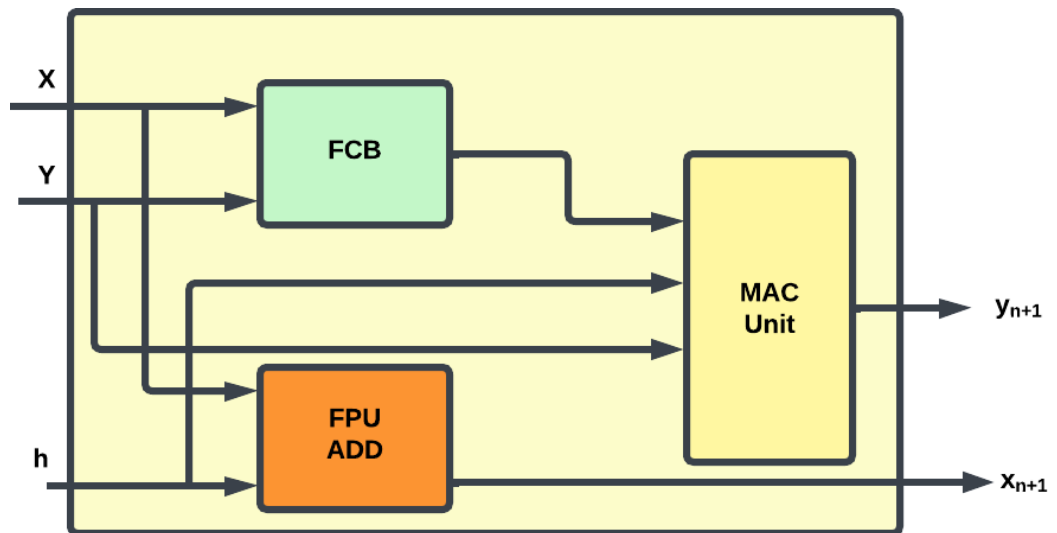


Figure 16

The Micro-Architecture Design Of The Euler Method Module



7.3.2 The Modified Euler Method Module

Two functional units have been implemented, apart from the Functional computation block (FCB) in our case. The first unit demonstrates the function $f(x_n, y_n)$, which does the specific operation in the equation (11). Meanwhile, another unit called the “Modified Euler Functional block” has also been implemented. Figure 17 displays the Modified Euler Functional block. Here, three single-precision floating-point IPs such as FPU Mul, MAC unit, and FPU_ADD have been utilized. With the step size ‘h’, an input port ‘a’ is initialized with value 0.05 to execute the operation of $h/2$ in a 32-bit floating-point multiply unit which will do the in the equations (3) and (4) for the modified Euler numerical method. Figure 15 performs all operations associated with obtaining the functional output of $f(x_n, y_n)$ in the equations (3) and (4) similarly. Figure 18, the next unit in sequence, will do the operation for getting the approximation of the iterations for the modified Euler method for solving the equation (1). In addition, another MAC unit along with a floating-point addition block has been developed. Table 2, given below, finally provides the number of Xilinx Vivado IP single-precision floating-point IP support blocks essential to build the function in (11).

Figure 17

The Modified Euler Functional Block

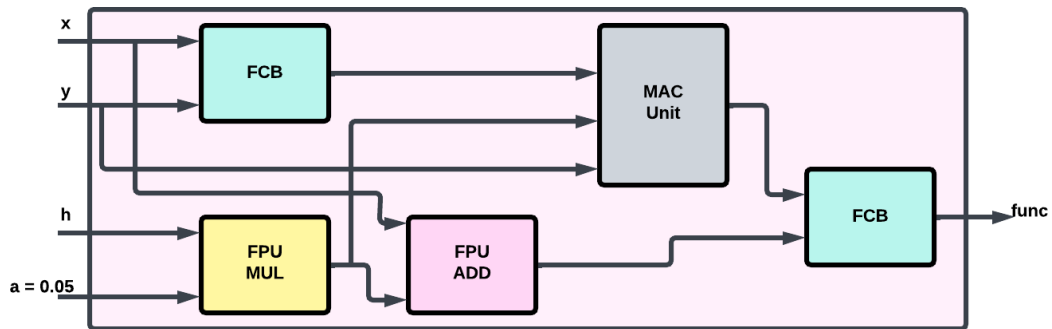


Figure 18

The Micro-Architecture Design Of The Modified Euler Method Module

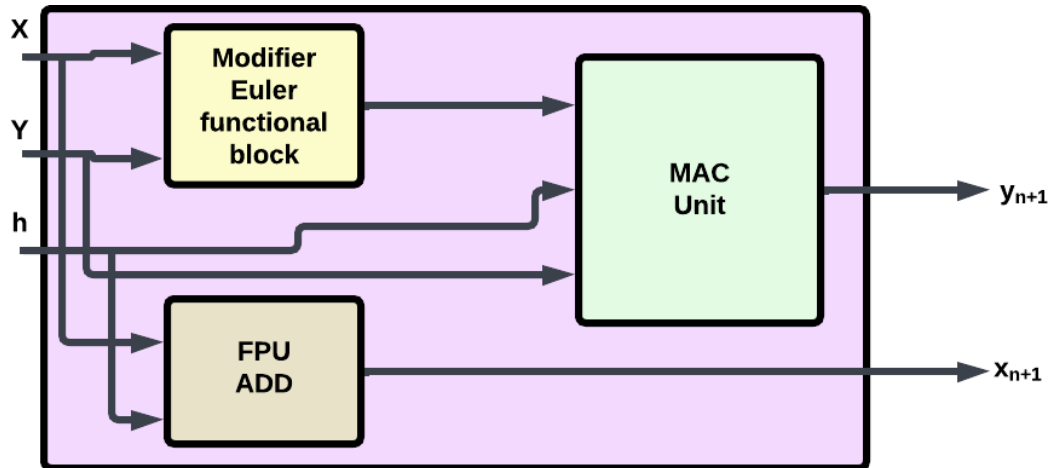


Table 2

The Number Of Xilinx Vivado IP Single-Precision Floating-Point IP Support Blocks Essential To Construct The Function In (11) Using The Euler And The Modified Euler Methods

| Modules | FPU_ADD | FPU_SUB | FPU_MUL | FPU_DIV | MAC Unit |
|-----------------------|----------------|----------------|----------------|----------------|-----------------|
| Euler method | 1 | 1 | 2 | 1 | 1 |
| Modified Euler method | 2 | 2 | 5 | 2 | 2 |

7.4 An Example Of One Set Of Inputs Being Processed By The Processor For The Euler Method Solver

Let us take equation (11) and solve it using the Euler method to demonstrate the flow of the architecture in designing the hardware accelerator.

According to the Euler method,

$y_{n+1} = y_n + hf(x_n, y_n)$, $x_{n+1} = x_n + h$, which are stated in equation (1) and (2). Our initial values of x_0 and y_0 are 2 and 1. The step size, h is 0.1. With these initial conditions, the first step is to derive the function specified in equation (11), which is $f(x, y) = xy(x - y)/2$. Therefore, $f(x_0, y_0) = x_0 y_0 (x_0 - y_0) / 2 = 2 \times 1 (2 - 1) / 2 = 2 \times 1 / 2 = 1$.

Therefore, for the 1st iteration, the value of $f(x_0, y_0) = 1$.

Now, for solving the function, f using Euler methods, we have to use this value in equation 1 to achieve the value of y_1 .

So, $y_1 = y_0 + hf(x_0, y_0) = 1 + (0.1 \times 1) = 1 + 0.1 = 1.1$.

Similarly, $x_1 = x_0 + h = 2 + 0.1 = 2.1$.

The results of 1st iteration to solve equation (11) using the Euler method are:

$y_1 = 1.1$. $x_1 = 2.1$.

For the second iteration, using the same procedure, the values of f , y_2 and x_2 have been derived.

$f(x_1, y_1) = f(2.1, 1.1) = 2.1 \times 1.1(2.1 - 1.1) / 2 = 2.31 / 2 = 1.155$. $y_2 = y_1 + hf(x_1, y_1) = 1.1 + (0.1 \times 1.155) = 1.1 + 0.1155 = 1.2155$.

$x_2 = x_1 + h = 2.1 + 0.1 = 2.2$.

Therefore, the final values of y_2 and x_2 after 2nd iteration are:

$y_2 = 1.2155$. $x_2 = 2.2$.

In this manner, we can solve first-order ordinary differential equations using the Euler method up to several iterations.

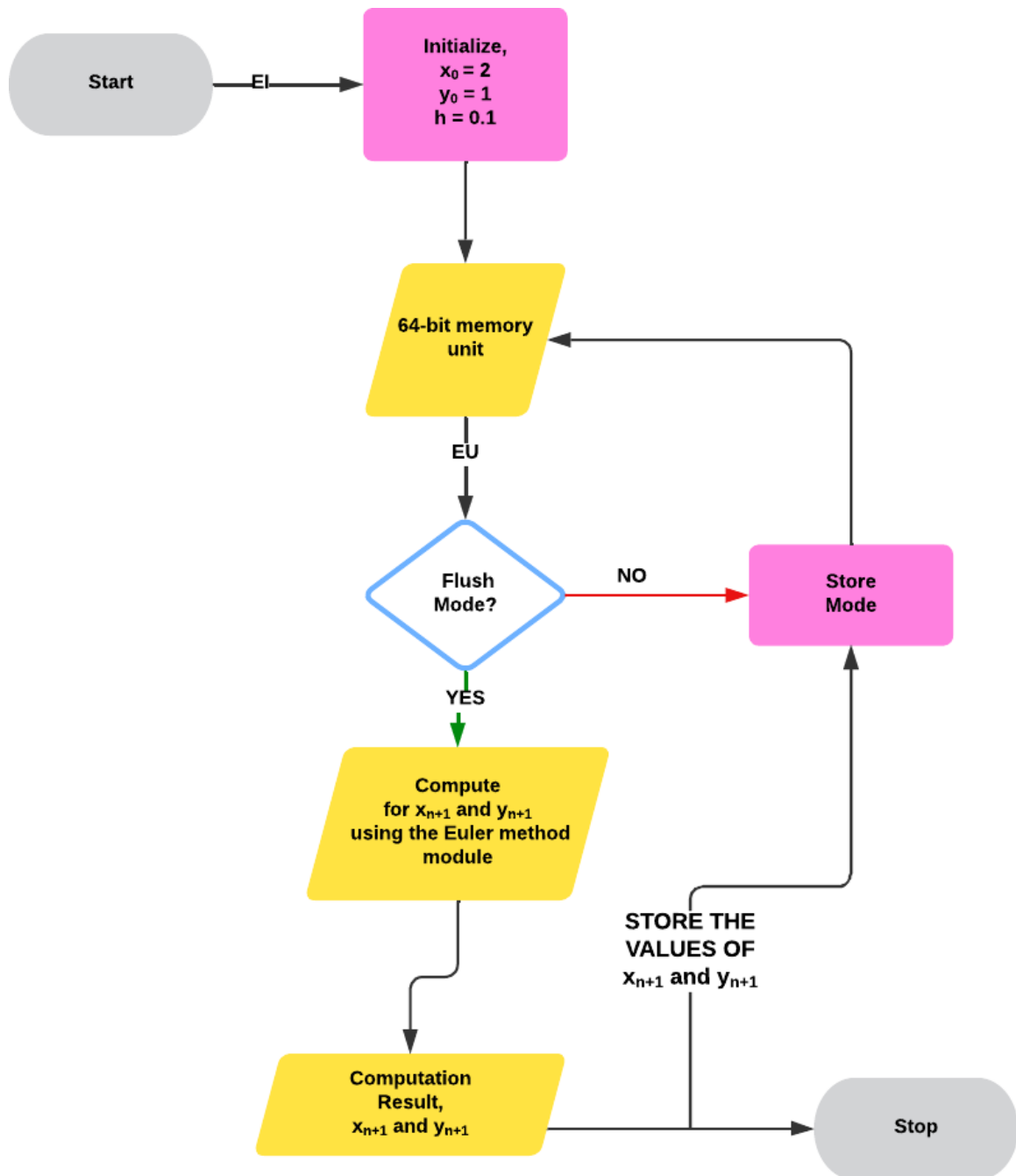
Now, as per the architecture described in Figure 14, whenever we are giving the

32-bit RISC-V-based “Euler initialization” instruction by the processor, as described in Section 7.2, the initial values of x_0 , y_0 , and h are 2, 1, and 0.1, respectively, according to our example, will be stored in three different source registers (rs1) of size 5 bits within the register file. These values will pass to the 64-bit memory unit as “contents.” The 32-bit RISC-V-based “Euler Update” instruction has two parts: flush and store modes, as described previously. The control unit will control the memory module through its “Flush” and “Store” mode. In the case of the Flush mode, when the “Flush” signal is equal to 1, the content of the memory unit will serve as inputs for the Euler method module, which has input ports of x , y , and h . The result from the functional computation block (FCB) in Figure 15 equates to $f(x_0, y_0)$, which is 1. This value then again passes through a MAC unit, specified in Figure 16, that performs the multiply-accumulate operation with the value of y_1 , which is 1 and $h = 0.1$. The final value of $y_1 = 1.1$ will be obtained from the Euler method module. Similarly, for $x_1 = 2.1$, the FPU_ADD unit in Figure 16, will add the value of $x_0 = 2$ and h , which is equal to 0.1, to get a result from the Euler method module.

In the Store mode, the values of $x_1 = 0.1$ and $y_1 = 1.1$ are then sent to the memory unit, which will be stored in new memory locations, and it flows to the Euler method module to do the operations for x_2 and y_2 . From the Euler method module, the values of $x_2 = 2.2$ and $y_2 = 1.2155$ will be generated. In this way, the micro-architecture will be able to generate the iterative solutions of the ODE using the Euler method through 32-bit RISC-V-based custom instructions from the processor. In Figure 19, a flowchart has been given to demonstrate the flow of operations for the Euler method in solving the ODE.

Figure 19

The Flow Diagram Of The Hardware Accelerator For The Euler Method In Solving The ODE



Chapter 8

Experimental Results

Due to the increased number of I/O pins on the ZYNQ - ZC702 FPGA Evaluation Board (xc7z020clg484- 1), the hardware accelerator has been synthesized to operate extremely well. The subsections include a detailed representation of the power consumption analysis, utilization summary, and timing analysis.

8.1 Power Analysis

The Xilinx Vivado Power Analyzer tool has been used to determine the total on-chip power consumption. Specifically, for both the Euler and modified Euler methods - operating at a clock frequency of 2.85Mhz, the total on-chip power consumed is typically identical when deployed on ZYNQ - ZC702 FPGA Evaluation Board (xc7z020clg484-1) with a slight increase of 0.001W in the modified Euler method, compared to its counterpart. For the Euler method, the total on-chip power consumed by the hardware accelerator is 0.191W, however utilizing the same parameters, the modified Euler method consumes the total on-chip power of 0.192W in respect to the single-precision floating-point operations. Also, the Euler method exhibits a dynamic power consumption of 1%, while the modified Euler method demonstrates 2%. In both cases, static sources consume the maximum power. In Figure 20, and Figure 21, the total on-chip power consumption by the hardware accelerator for both methods has been illustrated, using two pie charts.

In the case of the Runge-Kutta methods using the same micro-architecture in Fig.3 by just changing the computation unit and for equation 12, Figure 23 shows a graphical comparative study on power usage for all three floating point precision in Runge-Kutta

family numerical solutions, which has been designed in [56].

$$f = -2x - y \quad (12)$$

A design-space exploration has been given based on the on-chip power consumption of all three floating point formats for the Runge- Kutta solvers for ordinary differential equations in Figure 22, Figure 23, and Figure 24. Fourth-order Runge-Kutta (RK4) solver hardware accelerators use the highest power among three orders of Runge-Kutta equations in the following manner. For a half-precision format, RK4 needs 0.262W of power; for a single-precision format, it requires 0.635W, and for a double-precision format, it uses about 0.638W at an 80 MHz clock frequency. In all the other cases for power consumption, RK4 holds the maximum power in three floating point precision formats among all the other two orders of Runge-Kutta numerical solutions. Table 3 compares on-chip power consumption for equation (1) using second-order, third-order, and fourth-order numerical methods from the Runge-Kutta family in three floating-point precision formats and FPGA hardware resources respectively.

Figure 20

The Percentage Of The Total On-chip Power Consumption By The Euler Method In Solving The ODE In (11), Including The Static And Dynamic Sources

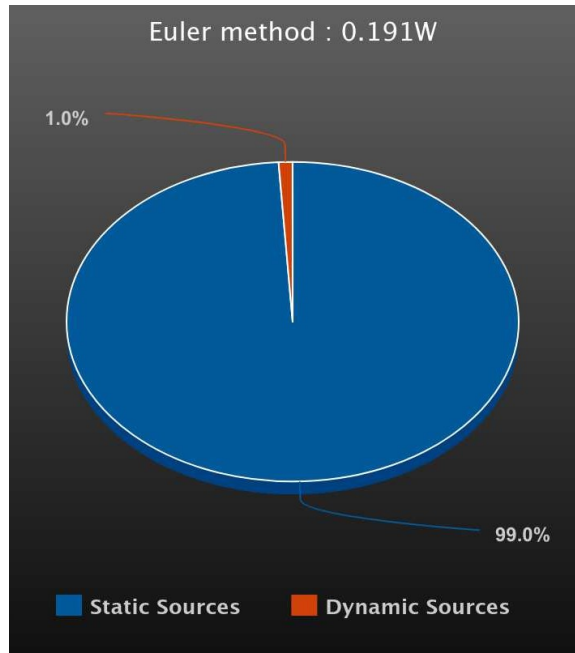


Figure 21

The Percentage Of The Total On-chip Power Consumption By The Modified Euler Method In Solving The ODE In (11), Including The Static And Dynamic Sources

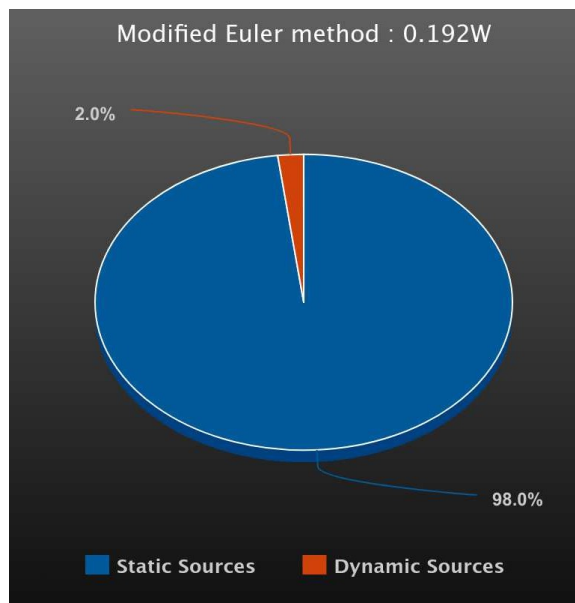


Figure 22

The On-chip Power Consumption Of Second-order, Third-order, And Fourth-order Runge-Kutta Family Methods For Equation (1) In Half-precision Floating-Point Format

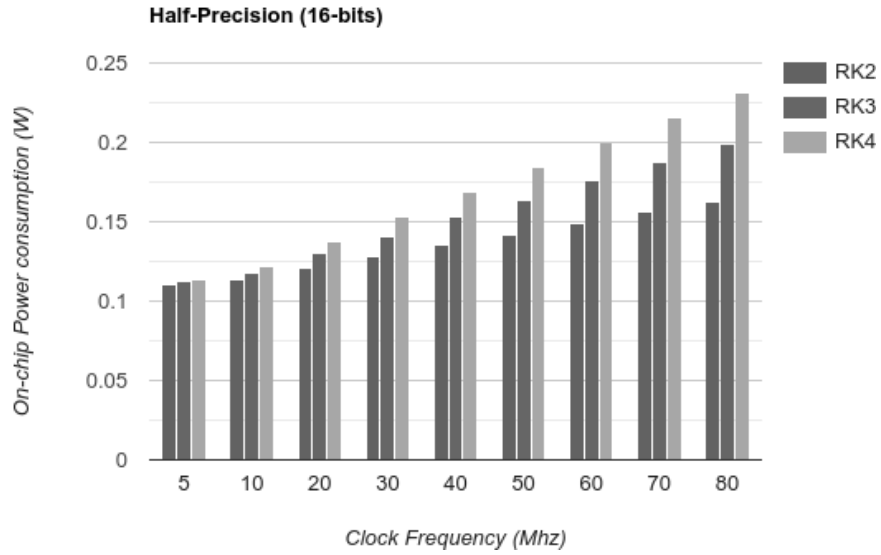


Figure 23

The On-Chip Power Consumption Of Second-Order, Third-Order, And Fourth-Order Runge-Kutta Family Methods For Equation (1) In Single-Precision Floating-Point Format

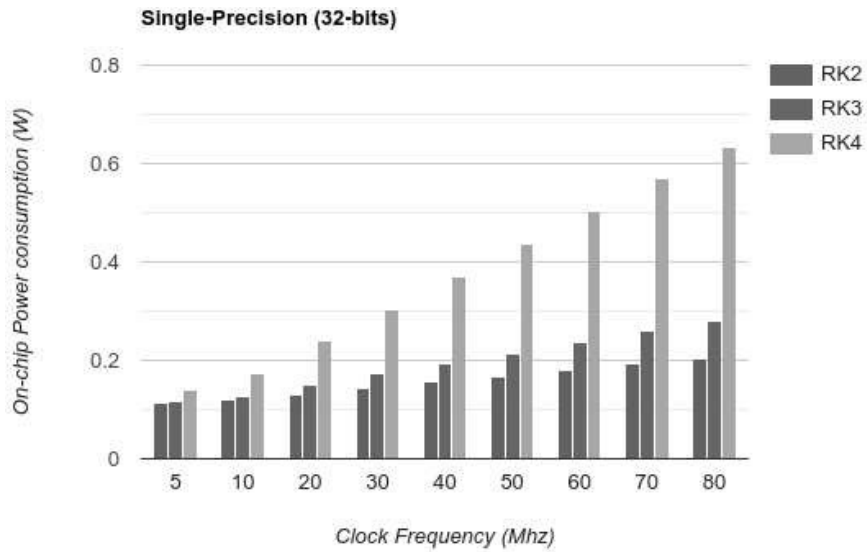


Figure 24

The On-chip Power Consumption Of Second-order, Third-order, And Fourth-order Runge-Kutta Family Methods For Equation (1) In Double-precision Floating-point Format

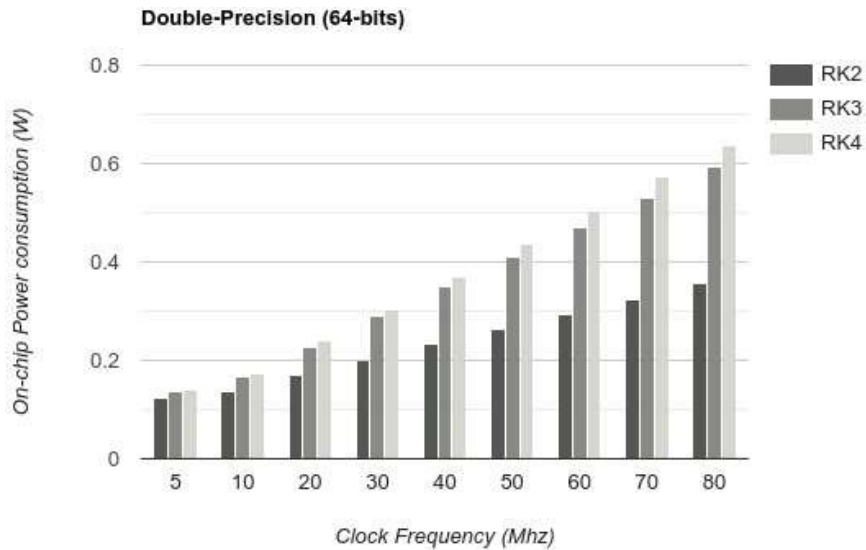


Table 3

Comparative Table Of On-Chip Power Consumption For Second-Order, Third-Order, And Fourth-Order Of Runge-Kutta Family Equations In Half, Single, And Double Floating-Point Precision Formats

| FP Precision Format | RK2 (Watts) | RK3 (Watts) | RK4 (Watts) |
|----------------------------|--------------------|--------------------|--------------------|
| Half (16-bits) | 0.165 | 0.199 | 0.231 |
| Single (32-bits) | 0.205 | 0.28 | 0.635 |
| Double (64-bits) | 0.356 | 0.592 | 0.638 |

8.2 Hardware Resource Utilization Summary

For each solver, the maximum number of look-up tables (LUTs), designs distributed RAM (LUTRAM), which means the LUTs can be used as synchronous RAM, flip-flops (FF), multipliers (DSP), input outputs (IO), high fanout buffer (BUFG), have been determined for the single-precision floating-point operations format. The resources are the estimation for implementing the accelerator for the ODE solvers. The results with the percentage of resources used in the FPGA are shown in Table 4 for each Euler and Modified Euler ODE solver. For every RK solver, the maximum number of look-up tables (LUTs), designs distributed RAM (LUTRAM), flip-flops(FF), multipliers (DSP), input outputs (IO), and high fanout buffer (BUFG) that can be used in half, single and double precision formats has been determined. Those results along with the percentage of resources utilized in the FPGA are displayed in Table 5 for ODE solver RK2, Table 6 for ODE solver RK3, and Table 7 for ODE solver RK4. These resources are a rough estimate to implement the accelerator for all three ODE solvers. We find in the tables the growing need for resources from single precision to double precision, and also between ODE solvers. Additionally, all resource needs are at their fullest maximum for RK4 in every precision format as expected.

Table 4

The Hardware Resource Utilization Summary Of The Accelerator Design With The Euler And The Modified Euler Methods In Solving The ODE In (11)

| Methods | BUFG (%) | IO(%) | DSP(%) | FFs(%) | LUTs(%) |
|----------------|----------|-------|--------|--------|---------|
| Euler | 3 | 71 | 5 | 2 | 9 |
| Modified Euler | 3 | 71 | 10 | 2 | 13 |

Table 5

FPGA Hardware Resources For Hardware Accelerator For The RK2 Method In All Floating-Point Precision Formats

| Elements | LUT | LUT- RAM | FF | DSP | IO | BUFG |
|-----------------|------------|---------------------|-----------|------------|-----------|-------------|
| RK2HP | 3% | 1% | 2% | 8% | 41% | 3% |
| RK2SP | 5% | 1% | 3% | 11% | 65% | 3% |
| RK2DP | 13% | 1% | 6% | 39% | 81% | 3% |

Table 6

FPGA Hardware Resources For Hardware Accelerator For The RK3 Method In All Floating-Point Precision Formats

| Elements | LUT | LUT- RAM | FF | DSP | IO | BUFG |
|-----------------|------------|---------------------|-----------|------------|-----------|-------------|
| RK3HP | 6% | 1% | 4% | 11% | 41% | 3% |
| RK3SP | 10% | 1% | 7% | 17% | 49% | 3% |
| RK3DP | 29% | 4% | 12% | 70% | 81% | 3% |

Table 7

FPGA Hardware Resources For Hardware Accelerator For The RK4 Method In All Floating-Point Precision Formats

| Elements | LUT | LUT- RAM | FF | DSP | IO | BUFG |
|-----------------|------------|---------------------|-----------|------------|-----------|-------------|
| RK4HP | 7% | 1% | 5% | 12% | 41% | 3% |
| RK4SP | 12% | 1% | 8% | 18% | 49% | 3% |
| RK4DP | 33% | 4% | 13% | 72% | 81% | 3% |

8.3 Timing Summary

The Euler method exhibits a total setup time delay of 6.832ns and a total hold time delay of 2.705ns. In comparison, the modified Euler method demonstrates a significantly higher setup time delay at 336.740 ns with only marginally reduced hold-time lag recorded as 2.230 ns. Considering a clock frequency operating at 2.85 ns, the worst negative slack (WNS) and worst hold slack (WHS) with the Euler method for solving the equation (11) are 88.016 ns and 0.165 ns. Employing identical parameters, the modified Euler method displays the worst negative slack of 13.827 ns and the worst hold slack is 0.183 ns. From the experimental results, it has been identified that the accelerator designed for the Euler method, has a lower setup time delay and hold time delay with higher worst negative slack and slightly lower worst hold slack. In the case of the accelerator designed for the modified Euler method, it has a significantly higher setup time delay with a marginally reduced hold time delay. Also, it exhibits a lower worst negative slack and slightly higher worst hold slack compared to the Euler method.

All the precision formats for every RK solver have determined the setup time, hold time, and pulse width at 80 MHz clock frequency. Table 8, Table 9, and Table 10 are given

to assess the timing summary of each solver.

Table 8

Timing Summary For The Hardware Accelerator For The RK2 Solver

| Elements | Setup Time(ns) | | Hold Time(ns) | | Pulse Width(ns) |
|----------|----------------|--------|---------------|-------|-----------------|
| | Min | Max | Min | Max | |
| RK2HP | 9.392 | 9.021 | 0.254 | 0.197 | 5.270 |
| RK2SP | 3.523 | 3.172 | 0.420 | 0.294 | 5.270 |
| RK2DP | 12.294 | 11.909 | 0.381 | 0.307 | 5.270 |

Table 9

Timing Summary For The Hardware Accelerator For The RK3 Solver

| Elements | Setup Time(ns) | | Hold Time(ns) | | Pulse Width(ns) |
|----------|----------------|--------|---------------|-------|-----------------|
| | Min | Max | Min | Max | |
| RK3HP | 7.663 | 2.147 | 0.425 | 0.310 | 4.020 |
| RK3SP | 3.924 | 3.562 | 0.418 | 0.286 | 6.520 |
| RK3DP | 11.626 | 11.316 | 0.412 | 0.243 | 5.270 |

Table 10

*Timing Summary For The Hardware Accelerator For The
RK4 Solver*

| Elements | Setup Time(ns) | | Hold Time(ns) | | Pulse Width(ns) |
|-----------------|-----------------------|--------|----------------------|-------|------------------------|
| | Min | Max | Min | Max | |
| RK4HP | 7.475 | 2.045 | 0.405 | 0.296 | 4.020 |
| RK4SP | 10.957 | 10.686 | 0.490 | 0.241 | 5.270 |
| RK4DP | 11.626 | 11.316 | 0.412 | 0.243 | 5.270 |

Chapter 9

Conclusion

Our preliminary research reveals the implementation of a hardware accelerator for the Euler and modified Euler numerical algorithms. The research highlights how important hardware acceleration is becoming in dealing with the computational needs of ordinary differential equations (ODEs) for different high-speed computing applications. Customized hardware accelerators for numerical methods like Euler and Modified Euler methods show a strong tactic to provide productive solutions for nonlinear equations. The use of Xilinx Vivado environment and VHDL highlights a strong setup to build hardware accelerators, showing promise for future progress in ODE-solving methods. It has been observed that the third-party vendor AXI4 stream Xilinx single-precision floating-point IP support significantly improves the hardware accelerator's performance. This breakthrough improves not only the existing implementation but also sets a base for making future accelerators that match various ODE solvers and floating-point representation units. Also, a thorough analysis, which includes power estimation, use of hardware resources, and timing summary has been given. The hardware accelerator will work effectively with the ZYNQ ZC702 FPGA Evaluation Board (xc7z020clg484-1) and at the moment, this article plays a vital role in improving the findings' capability to execute using alternate floating-point representations and methodologies for ODE solvers.

Chapter 10

Future Work

The future work will contribute to two subsections:

- Design-space exploration for Runge-Kutta Hardware Accelerator for a system of ordinary differential equations
- Hardware Acceleration for Multigrid and Numerical Methods: A Survey

10.1 Design-Space Exploration For Runge-Kutta Hardware Accelerator For A System Of Ordinary Differential Equations

A second-order ordinary differential equation can be expressed as a system of first-order differential equations. Generally, most of the numerical methods are designed to solve for first-order ODEs. Using this technique of breaking down the second-order ODEs into first-order ODEs will solve the equation faster and more efficiently using standardized numerical methods such as Euler methods, Runge-Kutta methods, and much more. This work will deal with the hardware implementation of the Runge-Kutta solver to effectively solve a system of equations with three variables. Furthermore, a design-space exploration will be done based on the system of equations [57].

10.2 Hardware Acceleration For Multigrid And Numerical Methods: A Survey

The intersection of micro-architectures and differential equations will be observed primarily in three different areas, as outlined below: 1. Hardware implementation of higher-order non-linear systems, such as chaotic systems, 2. Acceleration of scientific computing workloads, based on systems of differential equations, with commercial off-

the-shelf accelerators such as general-purpose graphics processing units, and 3. Acceleration of scientific computing workloads, based on systems of differential equations, with customized, application-specific hardware accelerators. Researchers often employ multigrid methods as preconditioners [58–60] and solvers [61]. These methods can be dissected into various modalities in any generalized application. One of those modalities is the grid formation algorithm. This algorithm can dictate the properties of the multigrid methods as structured, block-structured, or adaptively structured. Another way to classify these methods is based on computational performance and convergence trajectory. The path of dependency characterizes the multigrid methods into various cycle types such as V-cycle (a traversal pattern involving correction steps across different grid resolutions), F-cycle (Also known as 'Full Multigrid cycle' with an extra relaxation step at each grid level), and W-cycle (traversal pattern with additional correction steps at intermediate levels). This survey will examine the attributes of multigrid methods at the intersection of various modalities (grid formation algorithm, computational performance, convergence trajectory, and scope of parallelization). The survey will also delve into an overview of the current methodologies and implementations designed to accelerate several linear differential equations. This will provide a comprehensive understanding of how these technologies can be used to enhance the performance and efficiency of simulations in various scientific and engineering applications.

References

- [1] J. Von Neumann, "First draft of a report on the edvac," *IEEE Annals of the History of Computing*, vol. 15, no. 4, pp. 27–75, 1993.
- [2] J. Backus, "Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs," *Communications of the ACM*, vol. 21, no. 8, pp. 613–641, 1978.
- [3] Moore, G. Cramming more components onto integrated circuits. *Electronics* 38, 8 (Apr. 19, 1965), 56–59.
- [4] R. R. Schaller, "Moore's law: past, present and future," in *IEEE Spectrum*, vol. 34, no. 6, pp. 52-59, June 1997, doi: 10.1109/6.591665.
- [5] Moore, G. Progress in digital integrated electronics. In *Proceedings of the International Electronic Devices Meeting (Washington, D.C., Dec.)*. IEEE, New York, 1975, 11–13.
- [6] Streetman, Ben G.; Banerjee, Sanjay Kumar (2016). *Solid state electronic devices*. Boston: Pearson. p. 341.
- [7] Schauer, Bryan. "Multicore Processors – A Necessity" (PDF). Archived from the original (PDF) on 2011-11-25.
- [8] Khondker S. Hasan; John Antonio; Sridhar Radhakrishnan (February 2014). A New Composite CPU/Memory Model for Predicting Efficiency of Multi-core Processing. The 20th IEEE International Conference on High Performance Computer Architecture (HPCA-14) workshop. Orlando, FL, USA.
- [9] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach*, 6th ed. Cambridge, Mass: Morgan Kaufmann Publishers, an imprint of Elsevier, 2019, p. 540.
- [10] Dennis G. Zill (15 March 2012). *A First Course in Differential Equations with Modeling Applications*. Cengage Learning. ISBN 978-1-285-40110-2.
- [11] Icarus Verilog, "Icarus Verilog," [Online]. Available: <https://steveicarus.github.io/iverilog/>.
- [12] GTKWave, "GTKWave," [Online]. Available: <https://gtkwave.sourceforge.net/>.
- [13] YosysHQ, "Yosys," [Online]. Available: <https://github.com/YosysHQ/yosys>.

- [14] W. Haensch, E.J. Nowak, R.H. Dennard, P.M. Solomon, A. Bryant, O.H. Dokumaci, A. Kumar, X. Wang, J.B. Johnson, M.V. Fischetti, Silicon CMOS devices beyond scaling, *IBM J. Res. Dev.* 50 (4.5) (2006) 339–361.
- [15] M. Bohr, A 30-year retrospective on Dennard’s MOSFET scaling paper, *IEEE Solid-State Circuits Soc. Newslett.* 12 (1) (2007) 11–13.
- [16] H. Esmailzadeh et al., ”Dark Silicon and the End of Multicore Scaling,” in 2011 38th Annual International Symposium on Computer Architecture, pp. 365–376.
- [17] J. Cong, V. Sarkar, G. Reinman and A. Bui, ”Customizable Domain-Specific Computing,” in *IEEE Design Test of Computers*, vol. 28, no. 2, pp. 6-15, March-April 2011, doi: 10.1109/MDT.2010.141.
- [18] D.A. Patterson, J.L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, Elsevier Science, 2017.
- [19] M. Zahran, ”Heterogeneous computing: Here to stay,” *Queue*, vol. 14, no. 6, pp. 31–42, 2016.
- [20] M. Ravi, A. Sewa, S. T.G. and S. S. S. Sanagapati, ”FPGA as a Hardware Accelerator for Computation Intensive Maximum Likelihood Expectation Maximization Medical Image 14 Reconstruction Algorithm,” in *IEEE Access*, vol. 7, pp. 111727-111735, 2019.
- [21] A. Krishnakumar, U. Ogras, R. Marculescu, M. Kishinevsky, and T. Mudge, ”Domain-Specific Architectures: Research Problems and Promising Approaches,” *ACM Transactions on Embedded Computing Systems*, vol. 22, no. 2, Article 28, March 2023, pp. 1-26. DOI: 10.1145/3563946.
- [22] J. Cong et al., ”CHARM: A Composable Heterogeneous Accelerator-Rich Microprocessor,” in *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, pp. 379–384.
- [23] W.J. Dally, Y. Turakhia, and S. Han, ”Domain-Specific Hardware Accelerators,” *Communications of the ACM*, vol. 63, no.7, pp. 48–57, 2020.
- [24] B. Reagen, Y.S. Shao, G-Y. Wei, and D. Brooks, ”Quantifying Acceleration: Power/Performance Tradeoffs of Application Kernels in Hardware,” in *International Symposium on Low Power Electronics and Design (2013)*, pp. 395–400.
- [25] Patterson, D. A., & Hennessy, J. L. (2017). *Computer Organization and Design RISC-V Edition: The Hardware Software Interface (1st ed.)*. Morgan Kaufmann Publishers Inc.
- [26] ”The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2”, Editors Andrew Waterman and Krste Asanovi ´c, RISC-V Foundation, May 2017.

- [27] Weigel, M. (2012). Performance potential for simulating spin models on GPU. *Journal of Computational Physics*, 231(8), 3064-3082.
- [28] Andres, E., Widhalm, M., & Caloto, A. (2009, January). Achieving high-speed CFD simulations: Optimization, parallelization, and FPGA acceleration for the unstructured DLR TAU code. In *47th AIAA Aerospace Sciences Meeting including The New Horizons Forum and Aerospace Exposition* (p.759).
- [29] Gerhold, T. *Overview of the Hybrid Rans Code TAU*; Springer: Berlin/Heidelberg, Germany, 2005; Volume 89, pp. 81–92.
- [30] Corrigan, A., Camelli, F. F., Löhner, R., & Wallin, J. (2011). Running unstructured grid-based CFD solvers on modern graphics hardware. *International Journal for Numerical Methods in Fluids*, 66(2), 221-229.
- [31] Kroll, N., Rossow, C. C., Schwamborn, D., Becker, K., & Heller, G. M. (2002, September). A numerical flow simulation tool for transport aircraft design. In *ICAS 2002 Congress*.
- [32] Shi, Y., Green Jr, W. H., Wong, H. W., & Oluwole, O. O. (2011). Redesigning combustion modeling algorithms for the Graphics Processing Unit (GPU): Chemical kinetic rate evaluation and ordinary differential equation integration. *Combustion and Flame*, 158(5), 836-847.
- [33] Goodnight, N., Woolley, C., Lewin, G., Luebke, D., & Humphreys, G. (2005). A multigrid solver for boundary value problems using programmable graphics hardware. In *ACM SIGGRAPH 2005 Courses* (pp. 193-es).
- [34] Lin, Y., Wang, F., Zheng, X., Gao, H., & Zhang, L. (2013). Monte Carlo simulation of the Ising model on FPGA. *Journal of computational Physics*, 237, 224-234.
- [35] Sun, W., Wirthlin, M. J., & Neuendorffer, S. (2007). FPGA pipeline synthesis design exploration using module selection and resource sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2), 254-265.
- [36] Liu, I., Lee, E. A., Viele, M., Wang, G., & Andrade, H. (2012, April). A heterogeneous architecture for evaluating real-time one-dimensional computational fluid dynamics on FPGAs. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines* (pp. 125-132).IEEE.
- [37] Sano, K., Hatsuda, Y., & Yamamoto, S. (2011, June). Performance evaluation of FPGA-based custom accelerators for iterative linear-equation solvers. In *20th AIAA Computational Fluid Dynamics Conference* (p. 3223).
- [38] Ngouabo, U. G., & Tchamdjeu, F. X. N. (2022). FPGA implementation of nonlinear equations with delay, *Alexandria Engineering Journal*, 61(8), 6237-6246.
- [39] Inc, M. (2007). New L-stable method for numerical solutions of ordinary differential equations, *Applied mathematics and computation*, 188(1), 779-785.

- [40] Hollabough, A., & Chakraborty, D. (2022, May). An open-source co-processor for solving Lotka- Volterra equations. In 2022 IEEE International Symposium on Circuits and Systems (ISCAS) (pp. 1690-1694), IEEE.
- [41] Stamoulias, I., Möller, M., Miedema, R., Strydis, C., Kachris, C., & Soudris, D. (2017, June). High-performance hardware accelerators for solving ordinary differential equations. In Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (pp. 1-6).
- [42] Kasbah, S. J., Damaj, I. W., & Haraty, R. A. (2008). Multigrid solvers in reconfigurable hardware, *Journal of Computational and Applied Mathematics*, 213(1), 79-94.
- [43] B. Jamroz and P. Mullaney, "Performance of Parallel Sparse Matrix-Vector Multiplications in Linear Solves on Multiple GPUs," 2012 Symposium on Application Accelerators in High Performance Computing, Argonne, IL, USA, 2012, pp. 149-152.
- [44] Horowitz M. Computing's energy problem (and what we can do about it). In Proceedings of the IEEE International Solid-State Circuits Conference Digest of Technical Papers (San Francisco, CA, Feb. 9–13). IEEE Press, 2014, 10–14.
- [45] John L. Hennessy and David A. Patterson, "A new golden age for computer architecture," *Communications of the ACM*, vol. 62, no. 2, pp. 48-60, Feb. 2019.
- [46] D. Mira, E. J. Pérez-Sánchez, R. Borrell, and G. Houzeaux, "HPC-enabling technologies for high- fidelity combustion simulations," *Proceedings of the Combustion Institute*, vol. 39, no. 4, pp. 5091- 5125, 2023.
- [47] A. Auweter, A. Bode, M. Brehm, H. Huber, and D. Kranzlmüller, "Principles of Energy Efficiency in High Performance Computing," in *Information and Communication Technology for the Fight against Global Warming*, D. Kranzlmüller and A. Min Toja, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 18-25.
- [48] Soham Bhattacharya, "RISCV-HDP," [Online]. Available: <https://github.com/BhattSoham/RISCV-HDP/tree/main/week3>
- [49] Griffiths, D.F., Higham, D.J. (2010). Euler's Method. In: *Numerical Methods for Ordinary Differential Equations*. Springer Undergraduate Mathematics Series. Springer, London.
- [50] A. B. M. Hamed, I. Yuosif, I. A. Alrhaman, and I. Sani, "The accuracy of Euler and modified Euler technique for first order ordinary differential equations with initial condition," *American journal of Engineering Research (AJER)*, vol. 6, pp. 334–338, 2017.
- [51] P. Ravindra Babu, Usha Pal, R.N. Sen, R. Karthikeyan, Runge–Kutta type-2 method for solving reactor point kinetics equations and its validation by analyzing thermal and fast reactor benchmarks, *Annals of Nuclear Energy*, Volume 170, 2022, ISSN:0306-4549.

- [52] Faranak Rabiei and Fudziah Ismail, "Third-Order Improved Runge-Kutta Method for Solving Ordinary Differential Equation," *International Journal of Applied Physics and Mathematics*, vol. 1, no. 3, pp. 191-194, 2011.
- [53] J.C. Butcher, "A history of Runge-Kutta methods," *Applied Numerical Mathematics*, vol. 20, no.3, pp. 247-260, 1996.
- [54] García, Andrés, and Juan Andrés Roteta Lannes. "A generalized initial value problem for ODE's." *arXiv preprint arXiv:2206.05769* (2022).
- [55] A. Pimentel, "Exploring Exploration: A Tutorial Introduction to Embedded Systems Design Space Exploration," *IEEE Design & Test*, vol. 34, no. 1, pp. 77-90, Feb. 2017.
- [56] S. Bhattacharya and D. Chakraborty, "Design and Analysis of a Hardware Accelerator with FPU-Based Runge-Kutta Solvers," *2023 International Conference on Electrical, Communication and Computer Engineering (ICECCE)*, Dubai, United Arab Emirates, 2023, pp. 1-6, doi: 10.1109/ICECCE61019.2023.10442325.
- [57] R. Bronson and G. Costa, *Schaum's Outline of Differential Equations*, 4th Edition. New York, NY, USA: McGraw-Hill, 2014.
- [58] A. V. Knyazev and K. Neymeyr, "Efficient solution of symmetric eigen Value problems using multigrid preconditioners in the locally optimal block conjugate gradient method," *Electronic Transactions on Numerical Analysis*, vol. 15, pp. 38–55, 2003.
- [59] Xu, Jinchao. *Theory of multilevel methods*. Vol. 8924558. Ithaca, NY: Cornell University, 1989.
- [60] J. H. Bramble, J. E. Pasciak, and J. Xu, "Parallel multilevel preconditioners," *Mathematics of Computation*, vol. 55, no. 191, pp. 1–22, 1990.
- [61] J. Bolz, I. Farmer, E. Grinspun, and P. Schroeder. Sparse matrix solvers on the GPU: conjugate gradients and multigrid. In *ACM Transactions on Graphics (TOG)*, volume 22. ACM, 2003, 917–924.