if goingToCrash(): dont()
RISC-V Architecture for Motion Planning Algorithms in Autonomous UAVs

A senior design project submitted in partial fulfillment of the requirements for the degree of Bachelor of Science at Harvard University

Anthony J.W. Kenny
S.B. Candidate in Electrical Engineering

Faculty Advisor: Vijay Janapa Reddi

Harvard University School of Engineering and Applied Sciences
Cambridge, MA

March 1st, 2020
Version: 4.2

Abstract

This thesis describes a process for accelerating motion planning in autonomous robots through the design of specialised microarchitecture and instruction set architecture. First, it shows the analysis of computational performance of **RRT!** (**RRT!**), a sampling-based motion planning algorithm commonly used in autonomous drones. Having identified collision detection as the biggest area of opportunity for improved performance, it describes the process of designing specialized hardware, taking advantage of parallelization, that quickly detects collisions. Finally, it presents how this specialized functional unit can be implemented in a processor, and a RISC-V **ISA!** (**ISA!**) extension designed to massively reduce the execution time of collision detection.

Rewrite Abstract

Contents

List of Acronyms

First Use Application Programming Interface (API) Second Use API Plural Use APIs

Acronyms

API Application Programming Interface. v, *Glossary:* API

Glossary

Application Programming Interface (API): A particular set of rules and specifications that a software program can follow to access and make use of the services and resources provided by another particular software program that implements that API. v

Chapter 1

Introduction

1.1 Problem Summary

1.1.1 Background

The **UAV!** (**UAV!**) has been utilised in military applications extensively throughout the late 20th and early 21st century. However, over the last decade, their use in non-military uses, such as commerical, scientific, agricultural, and recreational, such that the number of civilian drones vastly outnumber military **UAV!**s. Particularly in the commercial sector, such rapid growth in the number and range of applications means that autonomy is key for the profitable adoption of **UAV!**s. Such autonomy relies on efficient computation of motion planning algorithms. However, the implementation of these algorithms can be quite computationally expensive, and thus slow and/or detrimentally power consuming. As such, this thesis aims to design specialized hardware to more efficiently compute motion plans for autonomous drones.

cite

Robotics

For well over 2000 years, the concept of robotics, albeit not always with such a term, has fascinated humans. As early as the first century A.D., the Greek mathematician and engineer, Heron of Alexandria, described more than 100 different machines and automata in *Pneumatica* and *Automata* [1]. In 1898, Nikola Tesla demonstrated the first radio-controlled vessel. Since then, the world has seen widespread application of robotics in manufacturing, mining, transport, exploration, and weaponry. For the last few decades, robots have operated in controlled, largely unchanging environments (e.g. an assembly line) where their environment and movements are largely known *a priori*.
However, in recent years a new generation of autonomous robots has been developed for a wide range of real-world, complex applications. The increasing trend the use of autonomous robots is shown in Figure 1.1. These new robots, unlike those traditional ones described

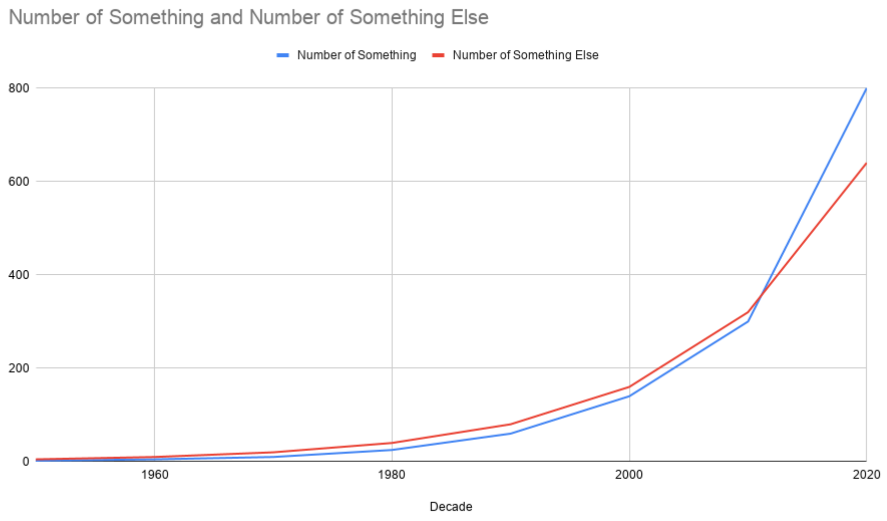


Figure 1.1: The use of Autonomous Robots over time [TODO]

above, are required to adapt to the changing environment in which they operate. As such, they must perform motion planning in real time.

Motion Planning

More of an introduction to motion planning.

Motion Planning refers to the problem of determining how a robot moves through a space to acheive a goal. Chapter 2 provides a detailed explanation of motion planning and of **RRT!**, a commonly used motion planning algorithm.

On the algorithmic level, motion planning has been extensively studied and many solutions exist. However, current algorithms running on regular **CPU!** (**CPU!**)s are too slow to execute in real time for robots operating in complex environments. Simply solving this problem with more raw computing power, using energy hungry **GPU!** (**GPU!**)s may have merit in tethered robots. On the other hand, untethered applications, such as autonomous drones, where limiting power consumption is a primary concern, this strategy is infeasible.

Subsubsection for Problem Overview: Add a subsubsection here about the current issue of slowness to transition into problem statement

1.1.2 Problem Definition

Problem Statement

Motion planning algorithms implemented in software that runs on general purpose **CPU!**s cannot execute quickly enough for fully autonomous **UAV!**s to operate in high-complexity environments. The state-of-the-art strategy of using power-hungry **GPU!**s to accelerate the execution of these algorithms requires too much power to be cost-effective or feasible for **UAV!**s to sustain flight for useful periods of time.

Improve Problem Statement: Existing research into accelerating robotic motion planning is <reason for RISC-V, inaccessable?> and mainly focussed on tethered arm moveing robots.

End User

This thesis aims to provide developers of autonomous drones with specialized hardware for motion planning. Such developers have a need for computing hardware that executes motion planning algorithms faster and more power efficiently than existing methods. This thesis will provide a processor design that is synthesizable on an **FPGA!** (**FPGA!**), giving developers a processer for which a **RTOS!** (**RTOS!**), or bare metal code, can be deployed.

Revise End User

1.2 Prior Work

1.2.1 Hardware Acceleration

Hardware acceleration refers to the strategy of using computer hardware specifically designed to execute a function more efficiently than can be achieved by software running on a general purpose **CPU!**. Specialized hardware designed to perform specific functions can yield significantly higher performance than software running on general purpose processors, and lower power consumption than **GPU!**s.

Computer Implementation Hierarchy

To briefly frame the space in which this thesis operates, consider the typical computer implementation hierarchy, demonstrated in Figure 1.2. **User level applications**, such as Google Chrome, Microsoft Word, and Apple's iTunes, sit at the top of the abstraction hierarchy. These applications are implemented in **High-Mid Level Languages**, such as C/C++, Python, Java, etc. These programming languages have their own hierarchy, but for the purpose of this thesis, it is sufficient to understand that these programming languages are then compiled into **Assembly Language**. Assembly language closely follows the execution of instructions on the **processor**, and is defined by an **ISA!**. An **ISA!** can be thought of as the contract between software programmers and processor engineers, agreeing what instructions the processor is able to implement. This assembly code is finally loaded into the processor's instruction memory and executed.

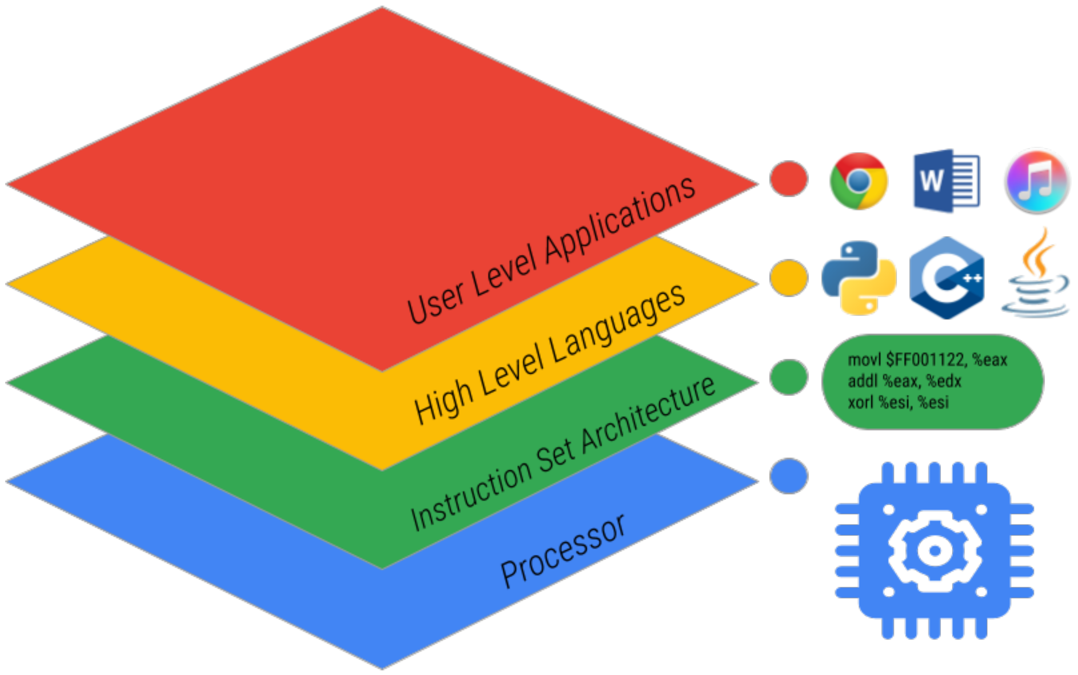


Figure 1.2: Simple Visualization of Computer Implementation Hierarchy

As will be outlined in Section 1.3, this thesis operates extensively on the lower two levels of this hierarchy, extending an existing **ISA!** and building hardware at the processor level that supports these extensions.

Acceleration of Motion Planning

Accelerating motion planning with hardware is a fairly well studied problem.
A Motion Planning Processor on Reconfigurable Hardware [2] studied the performance benefits of using **FPGA!**-based motion planning hardware as either a motion planning processor, co-processor, or collision detection chip. It targeted the feasibility checks of motion planning (largely collision detection) and found their solution could build a roadmap using the **PRM!** (**PRM!**) algorithm up to 25 times faster than a Pentium-4 3Ghz CPU could.
In *A Programmable Architecture for Robot Motion Planning Acceleration* [3], Murray et

al. built on the work of the aformentioned paper, to accelerate several aspects of motion planning in an efficent manner.
FPGA based Combinatorial Architecture for Parallelizing RRT [4] studies the possibility of building architecture to allow multiple **RRT!**s to work simultaneously to uniformly explore a map. Taking advantage of hardware parallelism allows systems such as this to compute more information per clock cycle.
Finally, in the paper *Robot Motion Planning on a Chip* [5], Murray et al. describe a method for contructing robot-specific hardware for motion planning, based on the method of constructing collision detection circuits for **PRM!** that are completely parallelised, such that edge collision computation performance is independent of the number of edges in the graph. With this method, they could compute motion plans for a 6-degree-of-freedom robot more than 3 orders of magnitude faster than previous methods.

1.2.2 RISC-V

RISC-V (pronounced "risk-five") is an **ISA!** developed by the University of California, Berkeley. It is established on the principles of a **RISC!** (**RISC!**), a class of instruction sets that allow a processor to have fewer **CPI!** (**CPI!**) than a **CISC!** (**CISC!**) (x86, the **ISA!** on which macOS and linux operating systems run, is an example of a **CISC!** instruction set). What makes RISC-V unique is its open-source nature. What makes **CPU!** design so expensive is that is requires expertise across many disciplines (compilers, digital logic, operating systems, etc). RISC-V was started with the philosophy of creating a practical, open-source **ISA!** that was usable in any hardware or software without royalites. The first report describing the RISC-V Instruction Set was published in 2011 by Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović [6].

Extending RISC-V

RISC-V is designed cleverly in a modular way, with a set of base instruction sets and a set of standard extensions. As a result, processors can be designed to only implement the instruction groups it requires, saving time, space and power on instructions that won't be used. In addition, another goal of RISC-V is to provide a basis for more specialized instruction-set extensions or more customized accelerators. This is described in the most recent *RISC-V Instruction Set Manual* [7]. This is a powerful feature, as it does not break any software compatability, but allows for designers to easily follow the steps outlined in Figure 1.3. From a hardware acceleration point of view, this is particularly useful as it allows the designer to directly invoke whatever functional unit or accelerator they implement from assembly code.

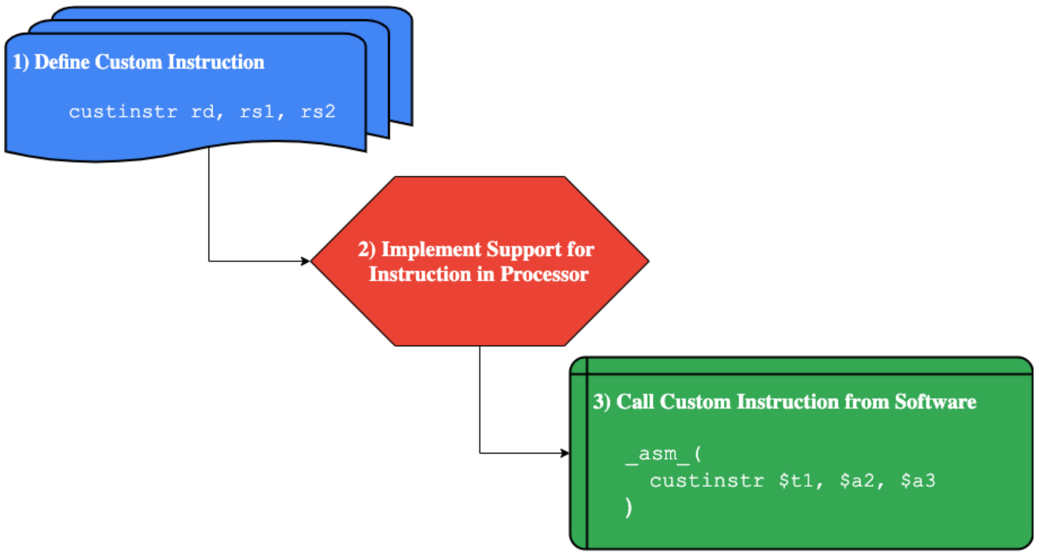

1) Define Custom Instruction
custinstr rd, rs1, rs2
2) Implement Support for Instruction in Processor
3) Call Custom Instruction from Software
asm(
custinstr $t1, $a2, $a3
)

Figure 1.3: Typical Process of Adding Non-Standard Extension to RISC-V ISA

Accelerating RISC-V Processors

Having only been released in 2011, RISC-V is still a relatively unexplored opportunity for non-education applications. However, it shows promise in the commercial space, with Alibaba recently developing the Xuantie, a 16-core, 2.5GHz processor, currently the fastest RISC-V processor. Recently there has been promising research into accelerating computationally complex applications, particularly in edge-computing, with RISC-V architecture. *Towards Deep Learning using TensorFlow Lite on RISC-V*, a paper co-written by the faculty advisor of this thesis, V.J. Reddi, presented the software infrastructure for optimizing the execution of neural network calculations by extending the RISC-V ISA and adding processor support for such extensions. A small number of instruction extensions achieved coverage over a wide variety of speech and vision application deep neural networks. Reddi et al. were able to achieve an 8 times speedup over a baseline implementation when using the extended instruction set. *GAP-8: A RISC-V SoC for AI at the Edge of the IoT* proposed a programmable RISC-V computing engine with 8-core and convolutional neural network accelerator for power efficient, battery operated, IoT edge-device computing with order-of-magnitude performance improvements with greater energy efficiency.

1.3 Project Overview

1.3.1 Proposed Solution

This thesis proposes a non-standard RISC-V Instruction Set Extension, supported by a functional unit embedded in a **FPGA!** synthesizable processor design that more rapidly computes motion planning for autonomous **UAV!**s. It will use the **RRT!** algorithm as a benchmark for performance analysis. Profiling of **RRT!** described in chapter 2 found that edge collision detection was the most performance limiting function of **RRT!**. As such, this thesis aims to design a RISC-V extension and specific circuitry that support the faster execution of edge collision detection.

System Overview

Figure 1.4 shows a high level overview of the system this thesis proposes.

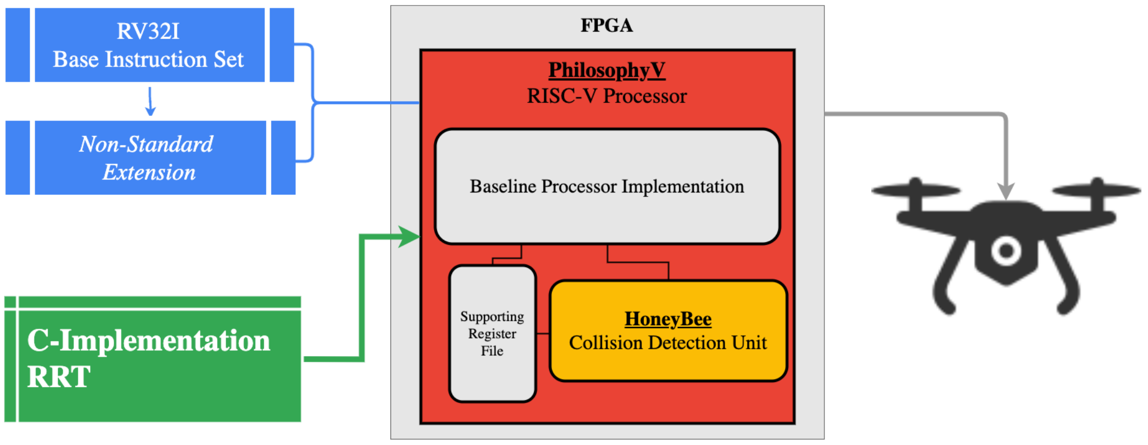


Figure 1.4: System Diagram of Overall Project

The **Extended RISC-V ISA!** is made up of the **RV32I!** (**RV32I!**) Base Instruction Set and a non-standard extension that this thesis will define. The **PhilosophyV Processor** is a RISC-V chip built in **HDL!** (**HDL!**) for this thesis. It implements both the **RV32I!** instruction set and the non-standard extension. The PhilosphyV Core includes, along with a baseline 5-stage processor implementation, the **HoneyBee** collision detection unit. A **C Implementation of RRT** is loaded into the instruction memory of the PhilosophyV processor. This processor, synthesized on an **FPGA!**, is used as the main processor, co-processor, or accelerator on an **Autonomous UAV!**. Table 1.1 outlines the components of this system and their descriptions.

Component	**Source**	**Description**
	RISC-V Instruction Set	
RV32I!	Berkeley	40 Instructions defined such that **RV32I!** is sufficient to form a compiler target and suport modern operating systems [7].
Extension	*New*	This is the custom extension defined by this thesis targeting motion planning instructions. It is outlined in Chapter 4.
	C-Implementation of RRT	
RRT	*New*	Due to lack of available implementations of **RRT!** suitable for the purposes of this thesis, **RRT!** was implemented from the ground up in C. This is detailed in Chapter 2
	FPGA Synthesized Chip	
Zynq-7000	Xilinx	The Zynq-7000 family of **SoC!** (**SoC!**)s are a low cost FPGA and **ARM!** (**ARM!**) combined unit.
PhilosophyV	*New*	The processor built for this thesis to demonstrate how the RISC-V extension and hardware unit work together. This is detailed in Chapter 4
HoneyBee	*New*	The functional unit designed specifically for faster execution of edge collision detection computations. Outlined in Chapter 3

Table 1.1: List of System Components and their Descriptions

1.3.2 Project Specifications

Project Specifications: These need significant revision from the last checkpoint

1.3.3 Project Structure

This report is structured to follow the timeline of this project, and is outlined below:

1. A benchmark motion planning algorithm, **RRT!**, was selected and implemented in software. Once implemented, a variety of performance analysis methods were used to profile the computational hotspots of the algorithm. It was found that edge collision detection was the critical function limiting execution time. This process is detailed in Chapter 2.

2. With edge collision detection having been identified as the critical function, the process of designing specialised hardware to execute this function began. The technical specifications, performance specifications, designs, build phases, measurement and analysis of this hardware unit is presented in Chapter 3.

3. With the aforementioned functional unit's performance verified in simulation, the next step was to implement this in a processor. First, a baseline processor was designed and built for this project to implement a base RISC-V instruction set. The performance of **RRT!** is again profiled on this baseline processor (as up until this point, it was profiled on x86 architecture). A non-standard extension to the RISC-V **ISA!** was then defined and support for this was implemented in the processor. Comparative performance analysis was then conducted. This process is described in detail in Chapter 4.

4. Chapter 5 is a discussion of results and future work.

Summary of Results: Do I need a summary of results section in the introduction?

Chapter 2

Motion Planning in Software

This thesis aims to design hardware that executes motion planning algorithms faster than those same algorithms can execute on generic hardware. Chatper 2 introduces motion planning and details the process of implementing and analysing **RRT!** to identify computational hot-spots in the algorithm and thus identify the biggest opportunities for hardware optimization.
Section 2.1 provides an introduction to Motion Planning Algorithms in general. Section 2.2 outlines the **RRT!** algorithm, and describes the implementation of **RRT!** for this project. Finally, Section 2.3 outlines a method for performance analysis of **RRT!** and the results of such analysis.

Write a better introduction that more accurately defines sections and makes the POINT of this chapter clear.

2.1 Background

Motion Planning Algorithms refer to the set of algorithms that find possible sequences of valid configurations for a robot in a space.

Background on Motion Planning Algorithms.: Need more background here. Needs more of a mathematical foundation

2.2 Rapidly-Exploring Random Tree

RRT! is an algorithm designed to efficiently search, and thus plan a path through, a high-complexity environment by randomly sampling points and building a tree. The algorithm randomly samples points, draws an edge from the nearest currently existing node in the tree, to grow the tree in the space. It is inherently biased to grow towards large unsearched areas of the problem. RRT was developed by S. LaVelle[8] and J. Kuffner[9]. It is used in autonomous robotic motion planning problems such as autonomous drones, the focus of this thesis.

2.2.1 Algorithm

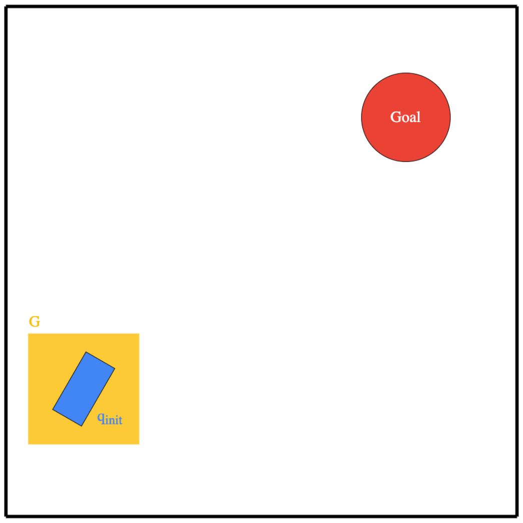
Building the Tree

Put simply, **RRT!** builds a tree (referred to as a graph) of possible configurations, connected by edges, for a robot of some physical description. It does so by randomly sampling the configuration space and adding configurations to the graph. From this graph, a path from the initial configuration to some goal configuration can be found, given a high enough number of iterations. As such, **RRT!** can be considered probabalistically complete. The pseudo-code for **RRT!** can be seen in Algorithm 2.1

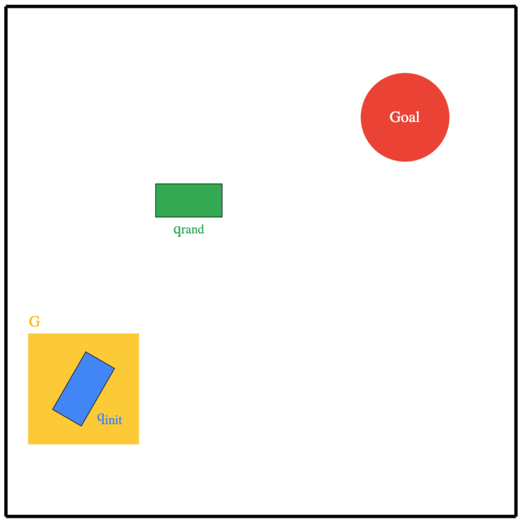
Algorithm 2.1: Rapidly-Exploring Random Tree in Free Configuration Space

```
Inputs:  Initial configuration $q_{init}$,
         Number of nodes in graph $K$,
         Incremental Distance $\Delta q$
Output:  RRT Graph $G$ with $K$ configurations $[q]$ & edges $[e]$
$G$.init();
for $k = 1$ to $K$ do
    $q_{rand} \leftarrow$ randomConfiguration();
    $q_{near} \leftarrow$ nearestVertex($q_{rand}$, $G$);
    $q_{new} \leftarrow$ newVertex($q_{near}$, $q_{rand}$, $\Delta q$);
    $G$.addVertex($q_{new}$);
    $G$.addEdge($q_{near}$, $q_{new}$);
end
```

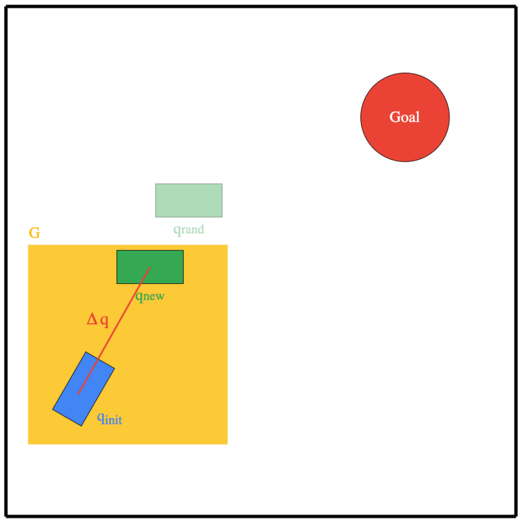
Algorithm 2.1 can be visually represented in Figure 2.1. Consider a **2D!** (**2D!**) robot operating in a **2D!** workspace. A Graph G is initialized containing an initial configuration, q_{init}, with constraints on the number of nodes that the graph can hold, K, and the maximum distance between two nodes, Δq. This is shown in Sub-figure 2.1a. A random configuration for the robot, q_{rand} is generated (2.1b). The nearest existing configuration in G, q_{near}, is found. (In the first iteration, $q_{near} = q_{init}$, shown in Sub-figure 2.1c). The distance between q_{near} and q_{rand} is calculated. If this distance is less than Δq, $q_{new} = q_{rand}$. If not, q_{new} is selected, typically by moving by Δq from q_{near} towards q_{rand} (2.1c). q_{new} is then added to G. This is repeated for K configurations.



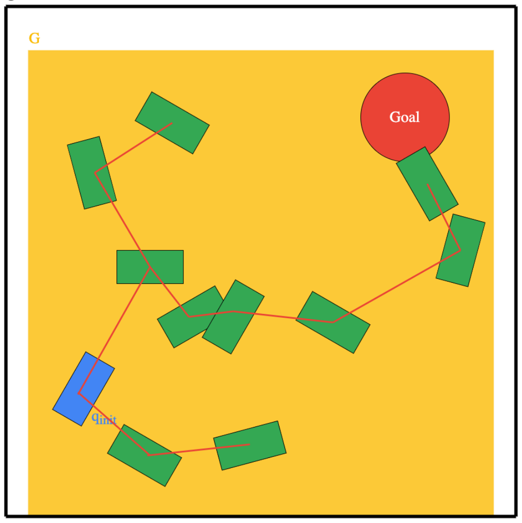
(a) Graph G contains only q_{init}



(b) The first random configuration, q_{rand}, is generated



(c) In first iteration, $q_{near} = q_{init}$. Distance between q_{init} and q_{rand} is greater than Δq, so q_{new} is generated and added to G



(d) This is repeated K times. For G, $K = 10$, and the red line represents the edges between configurations

Figure 2.1: Step by step demonstration of **RRT!** Algorithm for 2D robot in 2D space

Collision Detection

Algorithm 2.1 shows how **RRT!** builds a graph of possible configurations connected by edges in a free configuration space. However, in real-world applications, a robot's configuration space often contains obstacles. As such, collision detection must be included in the algorithm. The two types of collisions the algorithm must check for are *configuration collisions* (those where the robot would collide with an obstacle in a given configuration) and *edge collisions* (where the robot would collide when moving between two collision free configurations).

The RRT with configuration and edge collision detection can be seen in Algorithm 2.2. The method of implementing **RRT!** with collision detection to model a drone in 3D space is detailed in Section 2.2.2.

Algorithm 2.2: Rapidly-Exploring Random Tree with Collision Detection

```
Inputs:  Initial configuration $q_{init}$,
         Number of nodes in graph $K$,
         Incremental Distance $\Delta q$,
         Space $S$ containing obstacles
Output:  RRT Graph $G$ with $K$ configurations $[q]$ & edges $[e]$
$G$.init();
for $k = 1$ to $K$ do
    while !pointCollision($q_{new}$) do
        $q_{rand} \leftarrow$ randomConfiguration();
        $q_{near} \leftarrow$ nearestVertex($q_{rand}$, $G$);
        $q_{new} \leftarrow$ newVertex($q_{near}$, $q_{rand}$, $\Delta q$);
    end
    $e_{new} \leftarrow$ newEdge($q_{near}, q_{new}$)
    if !edgeCollision($e_{new}$) then
        $G$.addVertex($q_{new}$);
        $G$.addEdge($q_{near}$, $q_{new}$);
    else
        $k = k - 1$;
    end
end
```

2.2.2 Implementation

With **RRT!** selected as the benchmark algorithm against which to test specialised hardware, this project required an implementation of the algorithm that satisfied the following criteria.

Better RRT Implementation introductory sentence

Requirement	Description and Justification
C/C++ Implementation	As outlined in Section 1.3.3, the critical step in determining the design of specialized hardware to accelerate **RRT!** is CPU performance analysis of the algorithm to determine computational hot-spots. Implementations in C allow for the use of certain CPU profiling tools, described in Section 2.3.1, unlike higher-level languages such as Python.
Models Drone as Point	In reality, implementing **RRT!** for a drone would model the robot as a **3D!** (**3D!**) object defined by coordinates $\{x, y, z\}$ and Euler angles $\{\alpha, \beta, \gamma\}$. However, for simplicity's sake, modelling the drone as a point defined by coordinates $\{x, y, z\}$ will suffice. Time permitting, this could be revisited. Change this based on whether time does permit
Mirrors Algorithm	In order for the results of CPU performance analysis to be easy to understand, software implementation of **RRT!** should call functions that mirror the functions described in Algorithms 2.1 and 2.2.

Table 2.1: Technical Specifications for **RRT!** Implementation

Improve this table

The original intention was to find an existing implementation of RRT that could fulfill these requirements. Most open source implementations found online were in Python, and all those implemented in C were unsuitable[10][11][12][13], as they had extraneous **GUI!** (**GUI!**)s, reliance on external **API!** (**API!**)s, and other features that would distort analysis of algorithmic hot-spots.

As a result, it was necessary to build a C implementation of RRT from the ground up that satisfied the requirements in Table 2.1. It can be found in this project's GitHub repository. It follows Algorithm 2.1 closely. For monitoring correctness, I build in an optional **GUI!** that shows the tree, starting node, and obstacles.

Modelling a UAV! for RRT

Implementation in 2D

The first step was to implement RRT with a 2-Dimensional workspace.

More detail

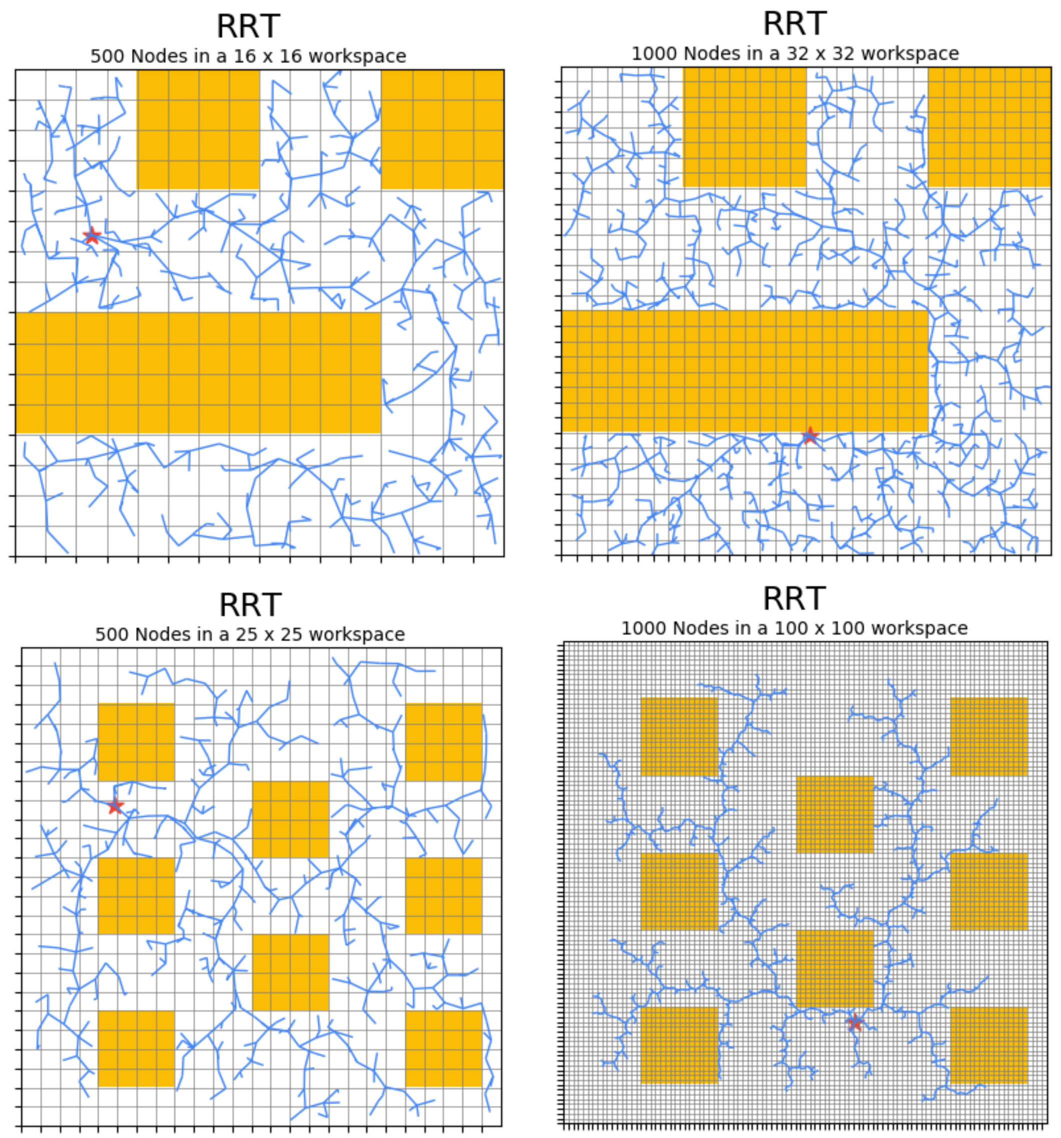


Figure 2.2: 2D RRT Implementation shown by **GUI!**

Implementation in 3D

Describe implementation in 3D

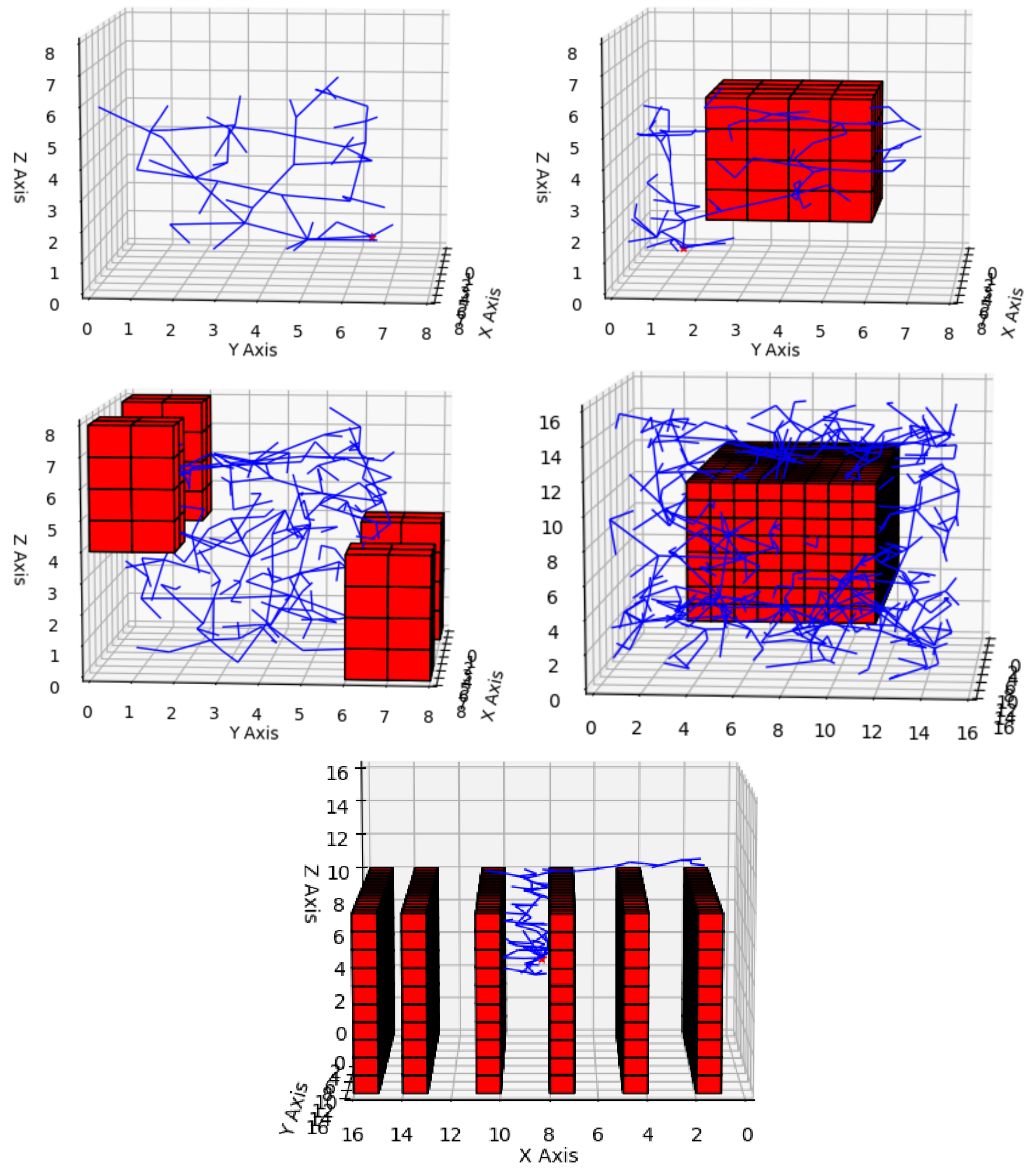


Figure 2.3: 3D RRT Implementation shown by **GUI!**

2.3 Performance Analysis

Brief introduction outlining purpose of performance analysis

2.3.1 Methodology

To restate, the aim of this thesis is to design a computer processor with reduced execution time of motion planning algorithms, such as **RRT!**. As such, it is important to understand the elements of the algorithm that have the highest percentage of CPU execution time. To determine this, it was necessary to implement my own, naive but typical, **RRT!** in C. This program could then be compiled and analysed using a software performance profiling tool. With this, I could design experiments to determine the critical RRT functions (those occupying a majority of CPU time) and see how this varies given different parameters.

Outline of method of analysis. Something better than the above

VTune Profiler

VTune Profiler performance profiler is an application for software performance analysis. It provides functionality to examine hot-spots for CPU execution time through a top down analysis, shown below in Figure 2.4. As can be seen from the figure, the top down analysis tool shows the percentage of CPU time taken up by each function. I used this tool to profile the algorithm's performance as I changed certain parameters.

Rewrite the above

Missing figure

Screenshot of VTune Top Down Analysis (Maybe)

Figure 2.4: VTune Amplifier TopDown Analysis Example

Internal Timing

The limitation of VTune Profiler is that it can only profile software running on Intel processors, which implement the x86-64 **ISA!**. As such, when the time comes to analyse performance of the software running on a RISC-V processor, another method will be required. A simple and effective way of measuring execution performance is to insert timing functionality into the software itself.

Provide or link to appendix of explanation of internal timing

Comparison

Before proceeding to use either of these methods to profile the software implementation of **RRT!**, it was important to verify that the two methods yielded similar results for the same program. Table 2.2 summarizes the results of analysis of a simple C executable. The program calls 5 functions, $\{A, B, C, D, E\}$, each a simple iteration in which a integer is incremented. Since the Internal Timing method returned similar results to the (trusted) VTune Profiler, it was considered to be a reliable method. While it was encouraging to see both methods returned similar results for absolute execution time, the more important metric was the similarity in percentage of total execution time.

function	Vtune Profiler		Internal Timing	
	time (s)	time (% total)	time (s)	time (% total)
A	0.488	57.4%	0.497	57.6%
B	0.2	23.5%	0.198	23.1%
C	0.102	12.0%	0.099	11.5%
D	0.048	5.7%	0.049	5.6%
E	0.012	1.4%	0.019	2.2%

Table 2.2: Comparison of Timing Methods

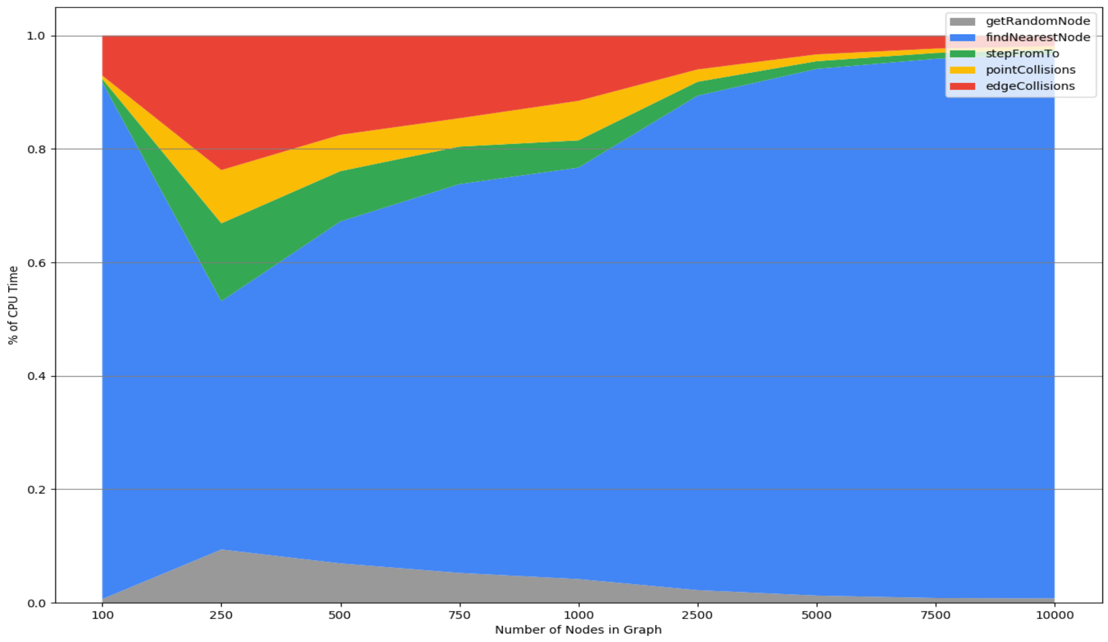
Experimental Design

In profiling **RRT!** in software, the goal was to find the critical task across different values of K and sizes of configuration space. Multiple tests were run, varying these two constraints, to find this critical function. The results of this analysis can be found in Section 2.3.2.

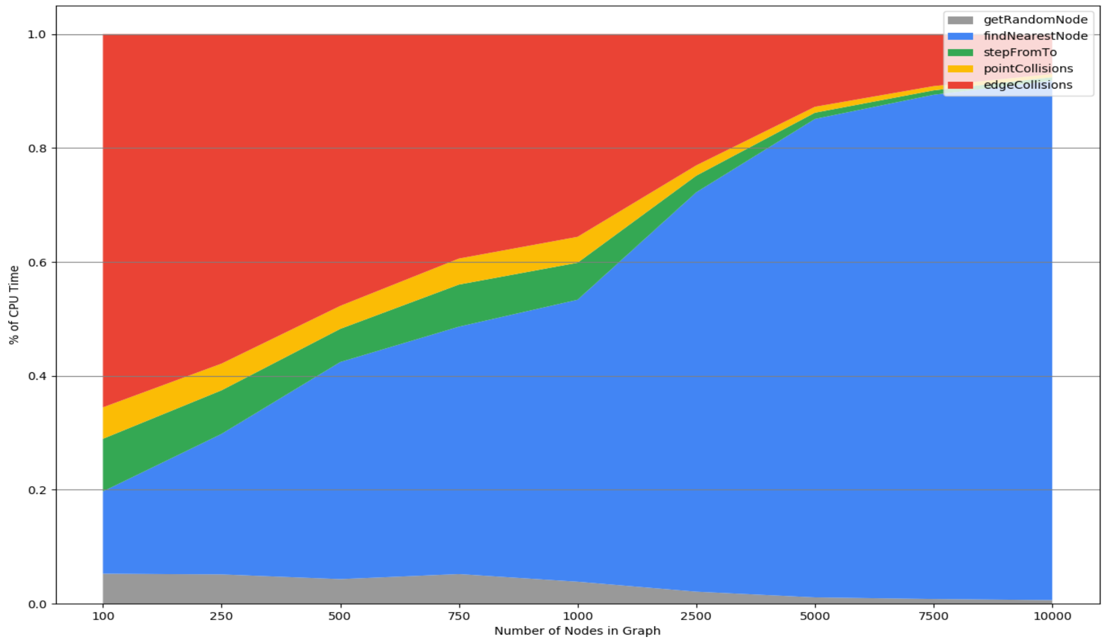
2.3.2 Results

Figure 2.5 shows the profile of functions within **RRT!**, for $100 \leq K \leq 10000$, and cubic configuration spaces with dimensions $\{4, 8, 16, 32\}$. Each subfigure shows a similar profile, with the % of CPU Execution Time taken by findNearestNode increasing with K. This is to be expected. However, it is also seen that edgeCollisions increases with larger configuration spaces, taking up the overwhelming majority of execution time for a 32x32x32 configuration space.

Explanation of time complexity

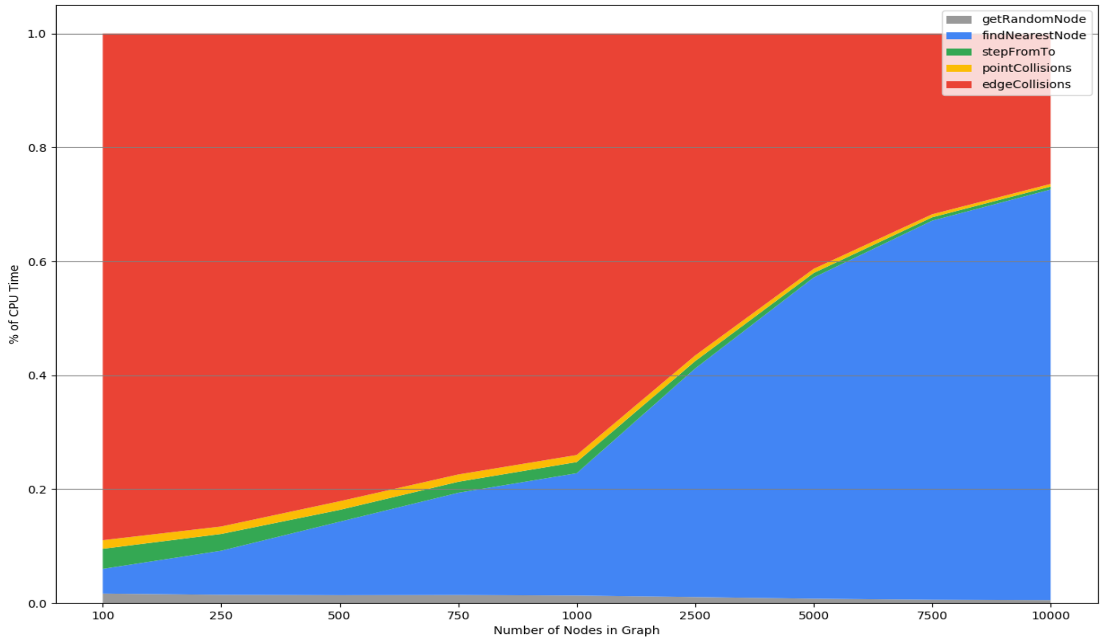


(a) 4x4x4 Configuration Space

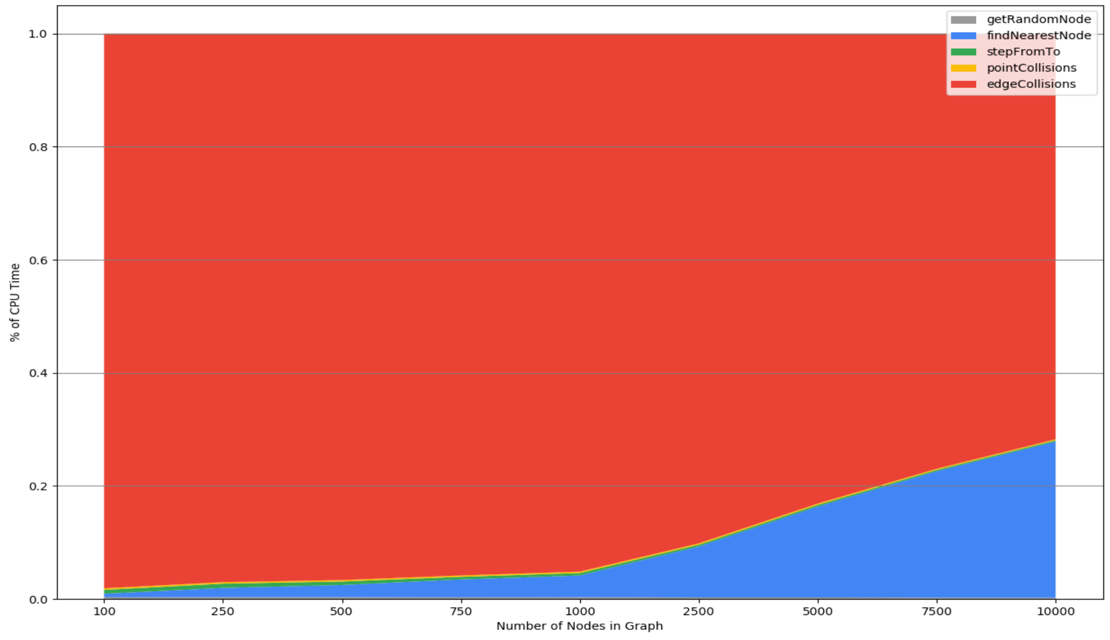


(b) 8x8x8 Configuration Space

Figure 2.5: **RRT!** Functions as a % of Total CPU Exectution Time



(c) 16x16x16 Configuration Space



(d) 32x32x32 Configuration Space

Figure 2.5: **RRT!** Functions as a % of Total CPU Exectution Time (cont.)

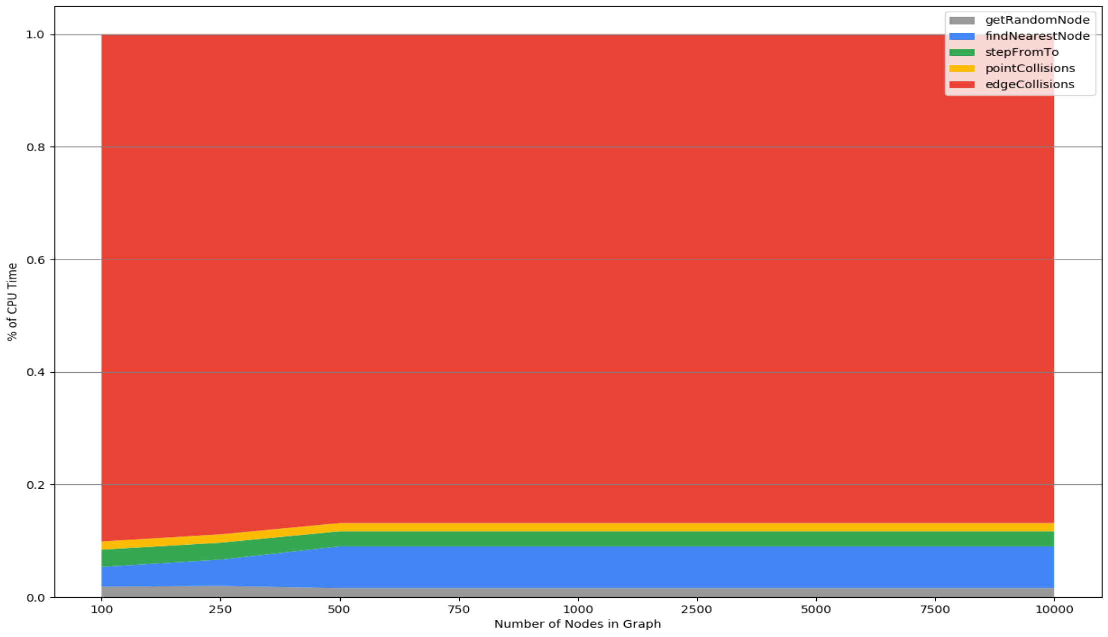
Change Y axis to % and increase text size

Furthermore, the computational load of findNearestNode can be reduced through a variety of software optimizations. A simple one used here to demonstrate that fact is storing nodes in seperate “buckets,” sorted by their x value. By using only two buckets, the execution time of findNearestNode fell drastically. Figure 2.6b shows edge collision detection accounting for over 95% of execution time for $100 \leq K \leq 10000$. This is consistent with the profiling results of **RRT!** in prior work[14].

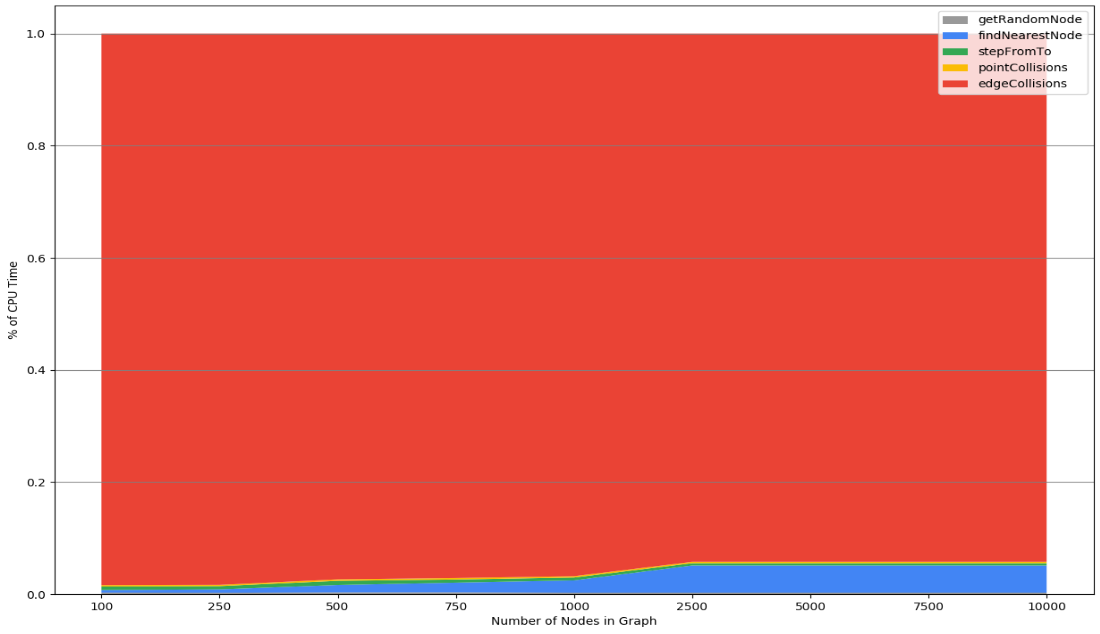
Conclusion

From the above data, it was identified that, as prior work suggested, edge collision detection shows the greatest promise for potential speedup through specialized hardware. The next chapter details the process of designing and building this hardware.

Add simulations to determine correct K



(a) 16x16x16 Configuration Space



(b) 32x32x32 Configuration Space

Figure 2.6: **RRT!** Functions Exectution Time, with Bucket Optimization

Chapter 3

Motion Planning in Hardware

3.1 Defining the Collision Detection Unit

It was demonstrated in Section 2.3.2 that the critical function of **RRT!** was edge collision detection. As such, the thesis proposes designing a functional unit that takes advantage of pipelining and parallelization to speed up the detection of edge collisions. Section 3.1.2 outlines the technical specifications for the functional unit. Section 3.1.3 outlines the performance specifications.

3.1.1 Edge Collision Function

Edge Collision Function Description: Edge collision function and algorithm, perhaps both for normal and parallelized?

3.1.2 Technical Specifications

Put simply, the functional unit that implements the edge collision detection function in hardware should have the same rough technical specifications as when the function is defined in software (Section 3.1.1). That is, it should take an edge e and an **OGM!** (**OGM!**) and return a boolean value: True, if the edge collides with an obstacle, otherwise False. Table 3.1 outlines the required technical specifications for the functional unit.

Element	**Description/Justification**
Contstraints	
Dimension N	N defines the dimension of the cubic configuration space for which the functional unit should take an identically sized **OGM!**
Inputs	
Edge e	An Edge e defined for a **3D!** configuration space by two points $\{p1, p2\}$, each defined by a set of **3D!** coordinates $\{x, y, z\}$.
Space S	In an abstract sense, the edge collision detection function takes Space S as an input. In a more practical sense, the functional unit will take an $N \times N \times N$ Occupancy Grid Map
Control Inputs	The functional unit must also have ports for control signals such as clock, reset, start, etc. These are required for adding the unit to a processor.
Outputs	
Return Value	1 bit return value: 1 if collision, 0 otherwise.
Control Outputs	Output ports for control signals such as idle, done, ready, etc. These are required for adding the unit to a processor.

Table 3.1: Technical Specifications for Edge Collision Detection Unit

Improve Technical Specifications: More detail on control units.

3.1.3 Performance Specifications

Performance Specifications Functional Unit

3.2 HoneyBee

The Honey Bee has long been renowned for its tireless work ethic. But people rarely give the Honey Bee credit for its remarkable navigation and collision avoidance strategies during flight. As such, it is quite appropriate that this functional unit, designed to work tirelessly, rapidly and efficiently to execute collision detection computations, is named HoneyBee.

More Iterations of HoneyBee Design: Note: Currently this report only shows the design/build/measurement of the first pass at designing the functional unit (Designated HoneyBee-A, or HB-A). Final report will detail further iterations.

3.2.1 Design

HoneyBee-A Design

The first design iteration, designated **HB-A!** (**HB-A!**), was designed to take advantage of the performance improvements associated with pipelining. Figure 3.1 demonstrates how pipelineing in hardware improves latency. By default, instructions are executed in order, one at a time. Pipelining takes advantage of operations that are independent of each other to reduce the number of clock cycles required to complete a set of instructions.

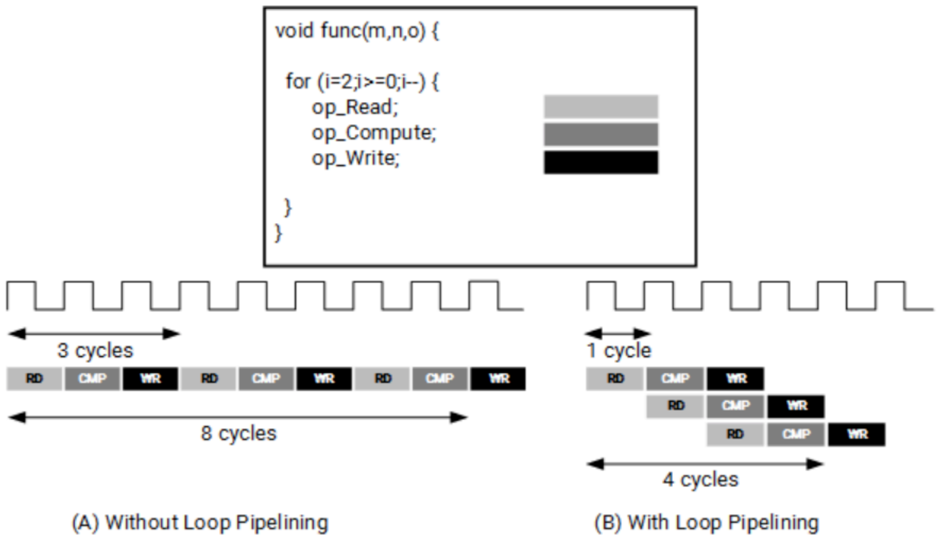


Figure 3.1: Pipelining to Improve Latency

Make my own version of this figure

3.2.2 Build

Hardware Description Languages

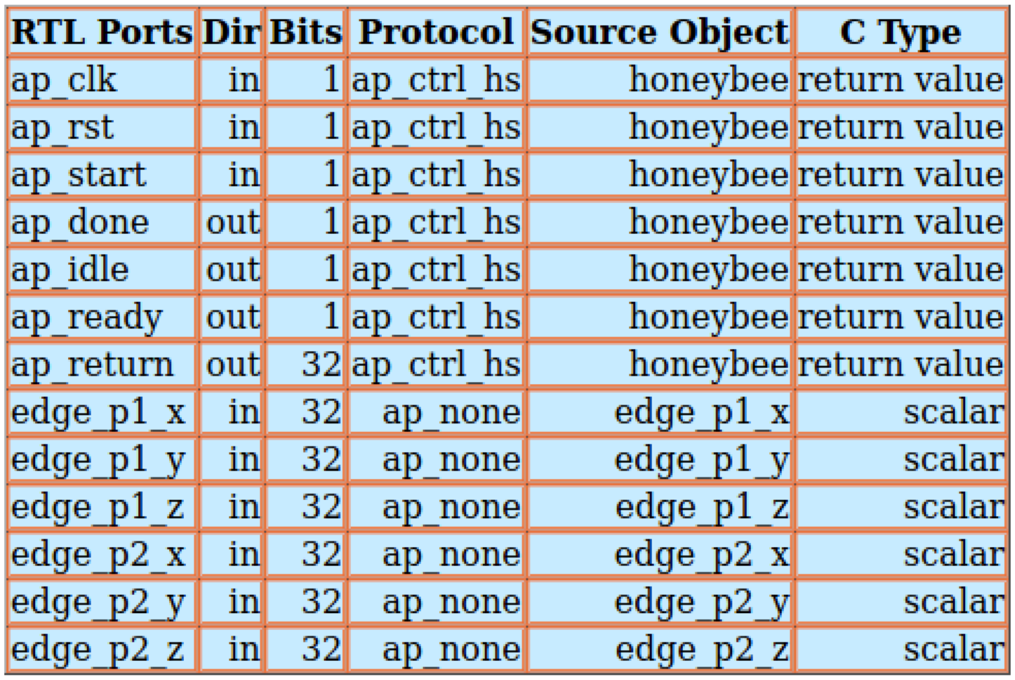
Introduction to Hardware Description Languages

High Level Synthesis

HLS! (**HLS!**) is an automated hardware design process that takes design files (written in high-level languages, such as C, C++ or SystemC) specifying the algorithmic function of a piece of hardware, interprets those files and creates digital hardware designs that execute this function. It effectively translates programming languages into hardware description languages. Key advantages of using HLS is speed and verification. It is much faster and easier to define functionality in C than it is in a **HDL!** such as Verilog, and thus design iterations are faster. It is also much simpler to verify one's design, as the functional units can be put through test benches written in C. This project used Vivado HLS to build the HoneyBee Unit.

HoneyBee-A Synthesis

Figure 3.2 shows the interface summary of successful synthesis of HoneyBee-A. Notice that the edge input has been split into 6 32 bit input ports.



RTL Ports	Dir	Bits	Protocol	Source Object	C Type
ap_clk	in	1	ap_ctrl_hs	honeybee	return value
ap_rst	in	1	ap_ctrl_hs	honeybee	return value
ap_start	in	1	ap_ctrl_hs	honeybee	return value
ap_done	out	1	ap_ctrl_hs	honeybee	return value
ap_idle	out	1	ap_ctrl_hs	honeybee	return value
ap_ready	out	1	ap_ctrl_hs	honeybee	return value
ap_return	out	32	ap_ctrl_hs	honeybee	return value
edge_p1_x	in	32	ap_none	edge_p1_x	scalar
edge_p1_y	in	32	ap_none	edge_p1_y	scalar
edge_p1_z	in	32	ap_none	edge_p1_z	scalar
edge_p2_x	in	32	ap_none	edge_p2_x	scalar
edge_p2_y	in	32	ap_none	edge_p2_y	scalar
edge_p2_z	in	32	ap_none	edge_p2_z	scalar

Figure 3.2: Interface Summary of HoneyBee-A Synthesis in Vivado HLS

Turn this into a table or get image from vivado

3.2.3 Measurement and Analysis

HoneyBee-A

The synthesis results of HoneyBee-A are shown in Table 3.2. It compares the execution time in microseconds for one edge to undergo collision detection if software and then in different synthesis “solutions”. MacOS and Ubuntu executing the function defined in honeybee.c have fairly similar results. Solution 1, which is the synthesized version of honeybee.c without any pipelining, was significantly slower. This is to be expected, as both MacOS and Ubuntu, operating on intel processors, would likely have some degree of pipelining and optimization of executing the compiled C code. However, significant improvements are observed once pipelining is implemented. Solutions 2-4 are increasing amounts of pipelining. Across the board, solutions 3 and 4 are roughly equal, but significantly faster that both solution 1 and the MacOS/Ubuntu execution times. Solution 4 shows a speedup of over 10x MacOS and Ubuntu.

Dimensions	Mac OS	Ubuntu	1	2	3	4
4x4x4	2	2	21.6	1.5	0.44	0.47
8x8x8	23	19	151	5.53	2.2	1.79
16x16x16	166	180	1133	41.37	13.08	12.11
32x32x32	1317	1424	8783	328	103	104

Table 3.2: Simulated performance of HB-A in microseconds

When HoneyBee-A is simulated in full **RRT!** execution, we see similarly promising results. Table 3.3 shows the results of simulated RRT execution with HoneyBee-A. This is also shown in Figure 3.3.

Executions	K	**Software**	**Sol. 1**	**Sol. 2**	**Sol. 3**	**Sol. 4**
1221	100	0.420	100.724	0.400	0.125	0.126
2986	250	1.251	26.226	0.979	0.307	0.310
5719	500	1.997	50.229	1.875	0.589	0.594
8299	750	2.907	72.890	2.722	0.854	0.863
11148	1000	4.798	97.912	3.656	1.148	1.159
27203	2500	9.172	238.923	8.922	2.801	2.829
54499	5000	18.509	478.664	17.875	5.613	5.667
80952	7500	27.833	711.001	26.552	8.338	8.419
107487	10000	36.311	944.058	35.255	11.071	11.178

Table 3.3: RRT Simulated Execution Times with HB-A (seconds)

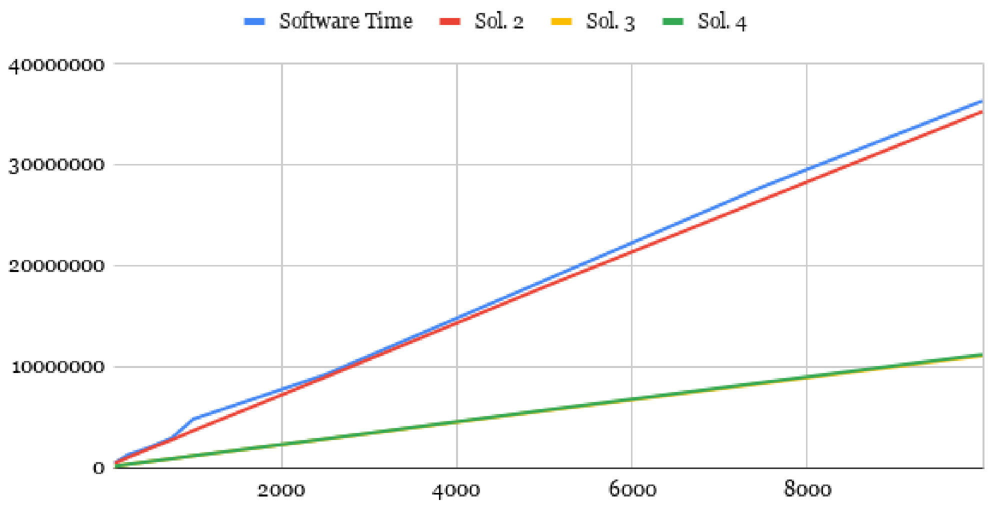


Figure 3.3: RRT Simulated Execution Time with HB-A (microseconds)

Make version of this chart in matplotlib for consistency and update axis

Expand Discussion of HoneyBee-A Results

Chapter 4

RISC-V Processor

4.1 Introduction to the Reduced Instruction Set Computer

4.1.1 Instruction Set Architecture

An Instruction Set Architecture can be thought of as an abstract model of a computer. On a broad level, it defines the data types, memory model, and registers of a computer, along with the instructions that it can execute.
It can also be thought as a “contract” between hardware and software developers. It is the promise made that the hardware will be able to execute all instructions defined in the **ISA!**, and the limitation that software must be compiled into that set of instructions.

4.1.2 RISC Processor Design

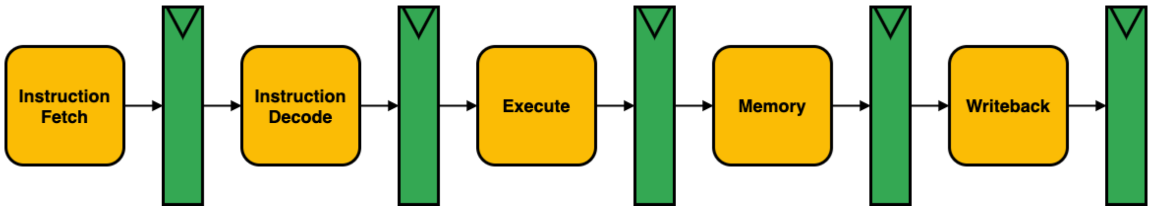


Figure 4.1: 5-Stage **RISC!** Datapath

4.2 RISC-V ISA

4.2.1 RV32I

The following is an excerpt from the RISC-V Specification, outlining the RV32I base integer instruction set [7]

> RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 40 unique instructions, though a simple implementation might ...[reduce] base instruction count to 38 total. RV32I can emulate almost any other ISA extension ...
> Subsets of the base integer ISA might be useful for pedagogical purposes, but the base has been defined such that there should be little incentive to subset a real hardware implementation ...

Registers

RV32I defines 32 unprivileged registers, each 32 bits wide. They are designated `x0-x31`, where `x0` is a hard-wired value of 0, and registers `x1-x31` hole values that various instructions use. RISC-V uses the load-store method, meaning that all operations perform on two registers or a register and an immediate, rather than performing operations directly on memory addresses. In addition, the 33rd unprivileged register is the program counter `pc`. Table 4.1 shows the register state for the RV32I Base Integer Instruction Set.

Register	ABI Name	Description
`x0`	`zero`	Hard-wired zero
`x1`	`ra`	Return address
`x2`	`sp`	Stack pointer
`x3`	`gp`	Global pointer
`x4`	`tp`	Thread pointer
`x5-7`	`t0-2`	Temporaries
`x8`	`s0/fp`	Saved register/Frame pointer
`x9`	`s1`	Saved register
`x10-11`	`a0-1`	Function arguments/return values
`x12-17`	`a2-7`	Function arguments
`x18-27`	`s2-11`	Saved registers
`x28-31`	`t3-6`	Temporaries
`pc`	`pc`	Program counter

Table 4.1: Register State for RV32I Base Instruction Set

Instruction Formats

Table 4.2 demonstrates the format of each different instruction type.

31	30 25	24 21	20	19 15	14 12	11 8	7	6 0	
funct7		rs2		rs1	funct3	rd		opcode	R-type
imm[11:0]				rs1	funct3	rd		opcode	I-type
imm[11:5]		rs2		rs1	funct3	imm[4:0]		opcode	S-type
imm[12]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Table 4.2: RV32I Base Instruction Formats

4.2.2 Motion Planning Extension

Motion Planning Extension: Full description of design of Non standard extension for motion planning. Should follow define, design, build, measure, analyse etc format.

4.3 PhilosophyV

Philosophy IV, written in 1903 by Mr. Owen Wister of the Class of 1882, recounts the antics of two Harvard students and their last minute attempts to study (or avoid studying) for an exam for which they are hopelessly unprepared. Similarly, this section details the process of my attempt to build a RISC-V processor, by far the most complex part of this Thesis, and a task for which I am unsure of my preparedness. As such, this processor is named Philosphy V; both in reference to the RISC-V ISA for which it is designed, and to the fact that my current situation seems much like a sequel to Mr. Wister's novel.

4.3.1 Baseline Implementation

Description of Baseline Philosophy V core

Figure 4.2 provides a schematic of the PhilosophyV processor.

Display bigger version of processor.

4.3.2 Implementing HoneyBee

Process of implementing honeybee into PhilV.

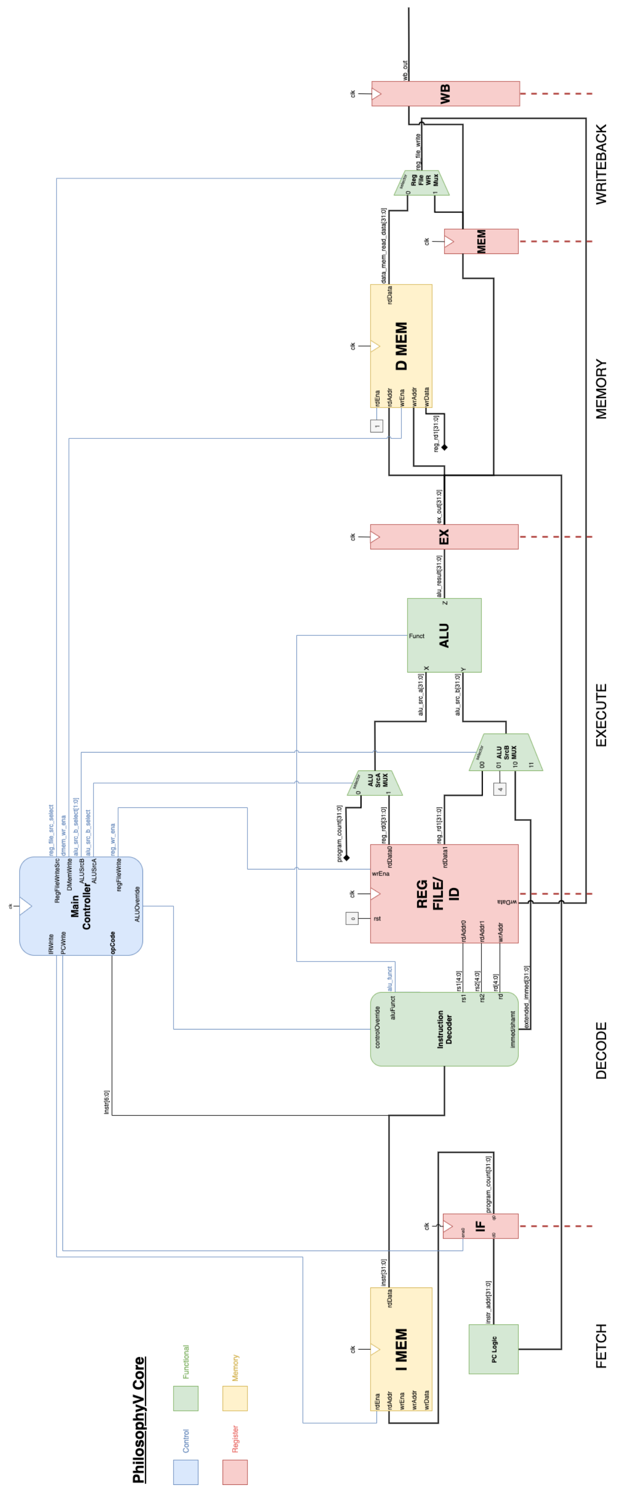


Figure 4.2: Philosophy V Processor

4.4 Performance Analysis

Comparative Performance Analysis of baseline and extended PhilV Core

Chapter 5

Conclusion

5.1 Discussion of Results

Discussion of Results

5.2 Evaluation of Success

Evaluation of Success

5.3 Future Work

Future Work

Bibliography

[1] H. (Alexandrinus), *De gli automati, overo machine se moventi, Volume 2.*

[2] N. Atay and B. Bayazit, “A motion planning processor on reconfigurable hardware,” in *Proceedings - IEEE International Conference on Robotics and Automation*, vol. 2006, pp. 125–132, 2006.

[3] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, “A Programmable Architecture for Robot Motion Planning Acceleration,” tech. rep.

[4] G. S. Malik, K. Gupta, K. M. Krishna, and S. R. Chowdhury, “FPGA based combinatorial architecture for parallelizing RRT,” in *2015 European Conference on Mobile Robots, ECMR 2015 - Proceedings*, Institute of Electrical and Electronics Engineers Inc., nov 2015.

[5] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, G. Konidaris, and D. Robotics, “Robot Motion Planning on a Chip,” tech. rep.

[6] V. I. B. U.-l. Isa, A. Waterman, Y. Lee, D. Patterson, K. Asanovi, and B. U.-l. Isa, “The RISC-V Instruction Set Manual v2.1,” *2012 IEEE International Conference on Industrial Technology, ICIT 2012, Proceedings*, vol. I, pp. 1–32, 2012.

[7] A. Waterman, K. Asanovic, and SiFive Inc, “The RISC-V Instruction Set Manual,” vol. Volume I:, 2019.

[8] S. M. LaValle, “Rapidly-Exploring Random Trees: A New Tool for Path Planning,” *In*, vol. 129, pp. 98–11, 1998.

[9] S. M. LaValle and J. J. Kuffner, “Randomized kinodynamic planning,” *International Journal of Robotics Research*, vol. 20, pp. 378–400, may 2001.

[10] RoboJackets, “RRT,” 2019.
https://github.com/RoboJackets/rrt.

[11] M. Planning, “rrt-algorithms,” 2019.
https://github.com/motion-planning/rrt-algorithms.

[12] Sourishg, “rrt-simulator,” 2017.
`https://github.com/sourishg/rrt-simulator`.

[13] Vss2sn, “Path Planning,” 2019.
`https://github.com/vss2sn/path{_}planning`.

[14] J. Bialkowski, S. Karaman, and E. Frazzoli, “Massively parallelizing the RRT and the RRT,” in *IEEE International Conference on Intelligent Robots and Systems*, pp. 3513–3518, 2011.

Appendices

Appendix A

Project Repository

This project's repository can be found at github.com/AnthonyKenny98/Thesis and contains mutliple subrepositories. It has the following structure.

Research

This folder holds the academic papers that constitute the background research of this Thesis.

Writeups

This folder holds the writeups required for this Thesis, including checkpoints in fulfillment of Harvard's ES100hf class and this Final Report

RRT

github.com/AnthonyKenny98/RRT
This subrepository holds both the 2D and 3D implementations of RRT used for this thesis, along with the tools required for both VTune Profiler and internal timing analysis.

HoneyBee

github.com/AnthonyKenny98/HoneyBee
This subrepository holds the HoneyBee functional unit, a hardware implementation of collision detection.

PhilosophyV

github.com/AnthonyKenny98/PhilosophyV
This subrepository holds the PhilosophyV RISCV chip

Appendix B

Budget

Budget

Appendix C

RRT Supporting Documentation

C.1 Justification of K:DIM Ratio

Todo list