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

Fix Cover Page Formatting

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 Rapidly-exploring Random Tree (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 Instruction Set Architecture (ISA) extension designed to massively reduce the execution time of collision detection.

Rewrite Abstract

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Introduction

Define goal: Somewhere here I need to define very clearly the following

1. Overall Goal:˜ Deliver an example of how RISC-V can be leveraged to design a custom processor for motion planning
2. 4 Objectives:
   - Profile a standard motion planning algorithm to determine the bottleneck function
   - Design a functional hardware unit that eliminates that bottleneck
   - Define a non standard extension
   - Implement Processor and verify all above

1.1 Problem Summary

1.1.1 Background

The Unmanned Aerial Vehicle (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 applications, such as commerical, scientific, agricultural, and recreational, has increased such that the number of civilian drones vastly outnumber military UAVs. 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 UAVs. 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

Autonomous 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* [**?**]. 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 complex applications. These new robots are required to adapt to the changing environments in which they operate; most often, this means planning their own paths through space. As such, they must perform motion planning in real-time.

Motion Planning

While most creatures in the animal kingdom find it relatively easy to navigate their surroundings, autonomous robots must be taught explicitly how to do so by their programmers. Motion Planning refers to the problem of algorithmically determining a collision-free path between two points in an obstacle-ridden space. Chapter **??** provides a detailed explanation of motion planning and of Rapidly-exploring Random Tree (RRT), a commonly used motion planning algorithm.

On the algorithmic and software level, motion planning has been extensively studied and optimized. Even so, current software implementations running on regular Central Processing Units(CPUs) are too slow to execute in real-time for robots to operate in rapidly chainging, high complexity environments. More powerful, highly parallelized Graphics Processing Units(GPUs) can be used in tethered robot applications (e.g. robotic arms autonomously executing pick-and-place functions). However, such GPUs consume far too much power to be used in autonomous drones, which are untethered and must sustain flight for useful periods of time. (A typical CPU uses between 65-85 watts, while some GPUs can use up to 270 watts).

cite

Application Specific Processors

Given the lacking performance in computing motion plans of a CPU, and the untenable power consumption of a GPU, autonomous drone developers are left with the option of developing an Application Specific Processor (ASP), optimized for motion planning.

However, designing a functional, high performance processor from scratch is no small task. It requires expertise in a variety of disciplines (compilers, digital logic, operating systems, etc), and an extradordinary amount of time and effort to develop and verify before it can be used. In short, it's an expensive process, which is why the market for computer

processors is dominated by companies like Intel, AMD, and ARM. The sharing of processor designs is also not possible, as commercial designs are proprietary and competing designs are not encouraged.

Finally, even if one were to design an ASP from scratch, or build off an existing commercial design (which means paying royalties), the Instruction Set Architecture (ISA) that the processor implements are not designed for are not designed for extendability, meaning that even a highly specialized processor is limited to a small number of instructions.

Discuss Moore's law and denard scaling

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 Reduced Instruction Set Computer (RISC), a class of instruction sets that allow a processor to have fewer Cycles Per Instruction (CPI) than a Complex Instruction Set Computer (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. 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 [**?**].

1.1.2 Problem Definition

Problem Statement

Motion planning algorithms implemented in software that runs on general purpose CPUs cannot execute quickly enough for fully autonomous UAVs to operate in high-complexity environments. The state-of-the-art strategy of using power-hungry GPUs to accelerate the execution of these algorithms requires too much power to be cost-effective or feasible for UAVs 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 Field Programmable Gate Array (FPGA), giving developers a processer for which a Real-Time Operating Systems (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 GPUs.

Computer Implementation Hierarchy

To briefly frame the space in which this thesis operates, consider the typical computer implementation hierarchy, demonstrated in Figure **??**. **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.

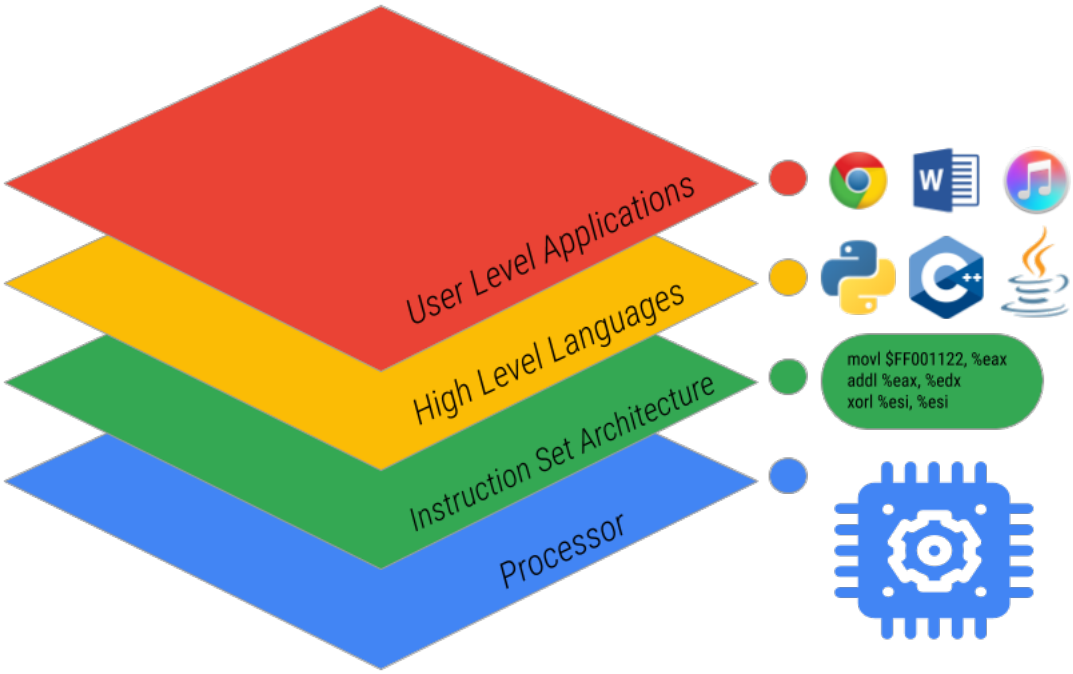


Figure 1.1: Simple Visualization of Computer Implementation Hierarchy

As will be outlined in Section **??**, 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 [**?**] 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 Probabalistic Road Map (PRM) algorithm up to 25 times faster than a Pentium-4 3Ghz CPU could.
In *A Programmable Architecture for Robot Motion Planning Acceleration* [**?**], 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 [**?**] studies the possibility of building architecture to allow multiple RRTs 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* [**?**], 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

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* [**?**]. 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 **??**. 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.

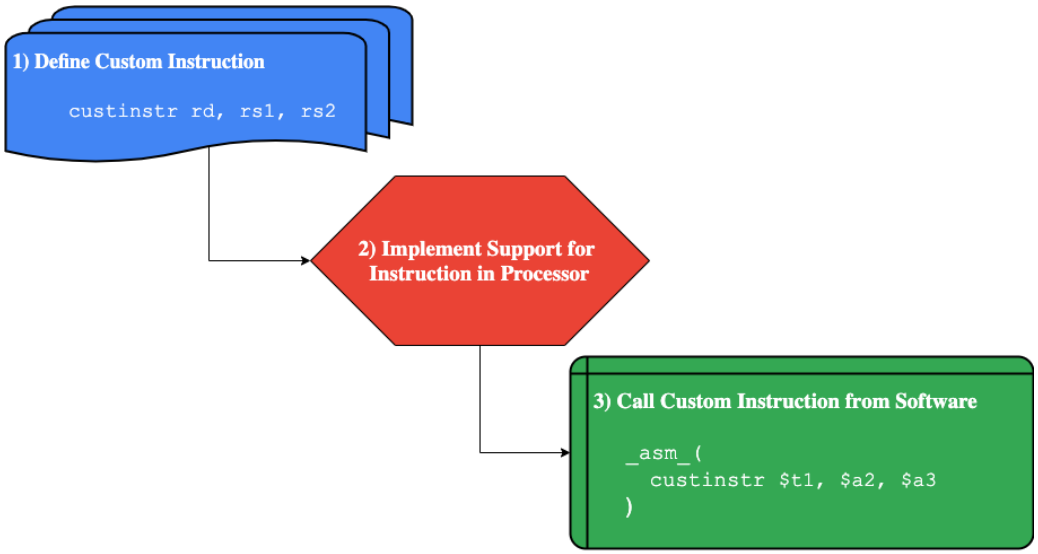


Figure 1.2: Typical Process of Adding Non-Standard Extension to RISC-V ISA

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 UAVs. It will use the RRT algorithm as a benchmark for performance analysis. Profiling of RRT described in chapter **??** 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 **??** shows a high level overview of the system this thesis proposes.

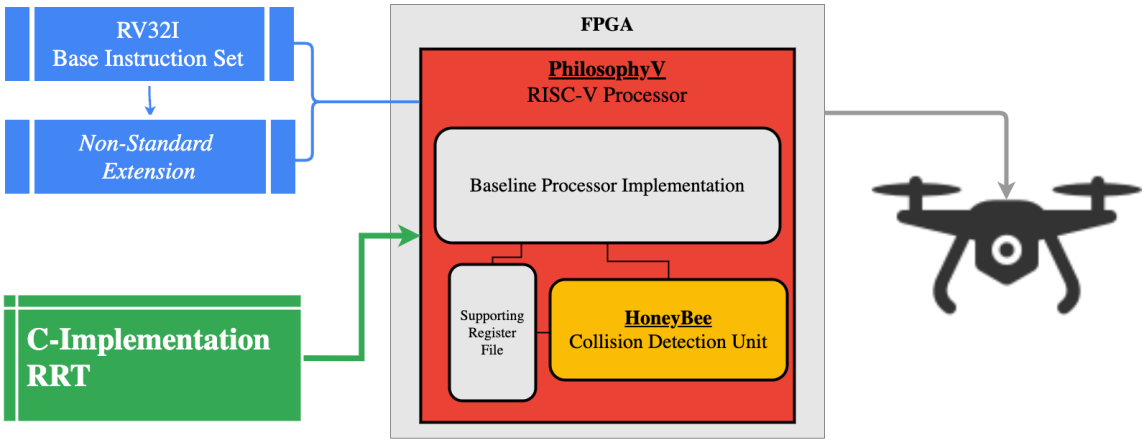


Figure 1.3: System Diagram of Overall Project

The **Extended RISC-V ISA** is made up of the RISC-V 32-Bit Integer (RV32I) Base Instruction Set and a non-standard extension that this thesis will define. The **PhilosophyV Processor** is a RISC-V chip built in Hardware Description Language (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 **??** 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 [**?**].
Extension	*New*	This is the custom extension defined by this thesis targeting motion planning instructions. It is outlined in Chapter **??**.
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 **??**
FPGA Synthesized Chip		
Zynq-7000	Xilinx	The Zynq-7000 family of System on Chip (SoC)s are a low cost FPGA and Advanced RISC Machine (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 **??**
HoneyBee	*New*	The functional unit designed specifically for faster execution of edge collision detection computations. Outlined in Chapter **??**

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

The first objective of this thesis is to identify a typical motion planning algorithm, profile its execution, and determine computational bottlenecks.

This chapter introduces the concept of motion planning and details the process of implementing and analyzing Rapidly-exploring Random Tree (RRT), a commonly used algorithm, to identify its computational bottlenecks.

Update once I have properly defined goals and objectives

2.1 Motion Planning Background

A funny paradox in computer science is the fact that it is relatively easy to teach a computer to perform tasks that humans find very complicated, but extremely difficult to program one to execute functions that humans master during infancy. Consider, it was as early as 1949 that Claude Shannon presented his paper *Programming a Computer for Playing Chess*[**?**], and by 1997 the *Deep Blue* computer defeated Garry Kasparov, the reigning world champion, in a six game chess match.[**?**] Compare that with some of the most advanced autonomous humanoid robots to date displaying dexterity only comparable with that of a toddler. The task of finding a collision free path, performed constantly without thought by a human, is an example of this paradigm. For a robot to plan its own paths, it relies on a set of Motion Planning Algorithms.

Motion Planning Algorithms refer to the set of algorithms that find possible sequences of valid configurations for a robot in a space. In more simple terms, they are algorithms that determine the movements a robot can make in a map, with the intent of eventually finding a path from one point to another.

2.1.1 Key Concepts

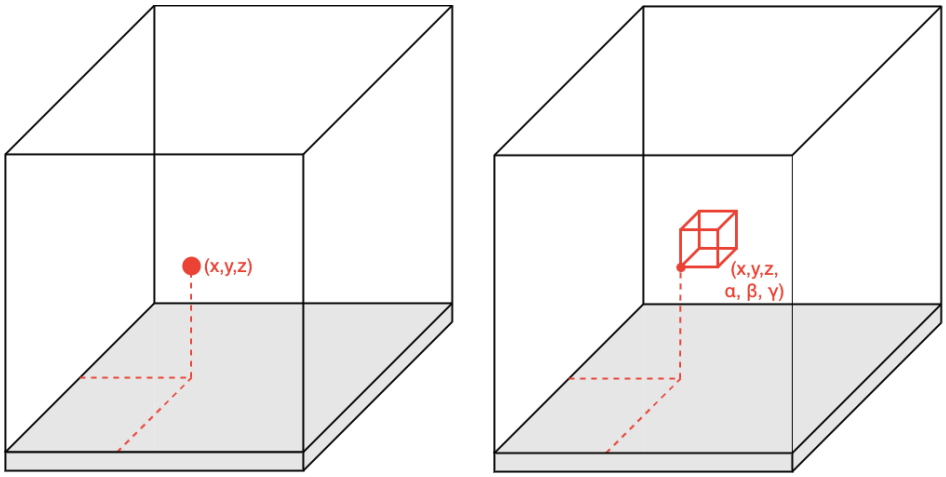
Workspace

The workspace, more loosely known as the **map**, is the space which the robot and obstacles occupy. Obviously, **obstacles** refer to anything with which the robot cannot intersect.

Configuration

A configuration describes the position, orientation, and pose of the robot. The complexity of a robot's configuration is therefore dependant on the dimension of the workspace, the complexity of the robot itself, and in what level of detail the robot must be represented. For example:

- Most simply, a robot can be represented as a point; by the Cartesian coordinates (x, y) in 2-Dimensional (2D) space and (x, y, z) in 3-Dimensional (3D) space.
- More realistically, a robot such as a drone may be represented in 3D as a 3D rectangular prism; by an origin point (x, y, z) and 3 Euler angles (α, β, γ) describing its orientation.
- In a more complex form, a fixed-base, N Degree-of-Freedom (DOF) robot would require an N-dimensional configuration.



(a) A robot represented by just a point in 3D space, requiring only 3 Cartesian coordinate (x, y, z) points to describe its configuration

(b) A robot represented as a cube in 3D space, now requiring 3 Euler angles (α, β, γ) along with the original Cartesian coordinates.

Figure 2.1: **Example of 2 Robot Configurations in 3D Space for Motion Planning Purposes**

Occupancy Grid Map

An Occupancy Grid Map (OGM) is a method of representing the obstacles present in a workspace. Obstacles are often irregularly shaped and computing collisions with such obstacles is near impossible. Therefore, the workspace is discretized into grids, with grids that contain any part of the obstacle marked as occupied, even if only a small part of the grid is occupied. An Occupancy Grid Map (OGM) will more accurately represent a workspace with a higher resolution, shown in Figure **??**.

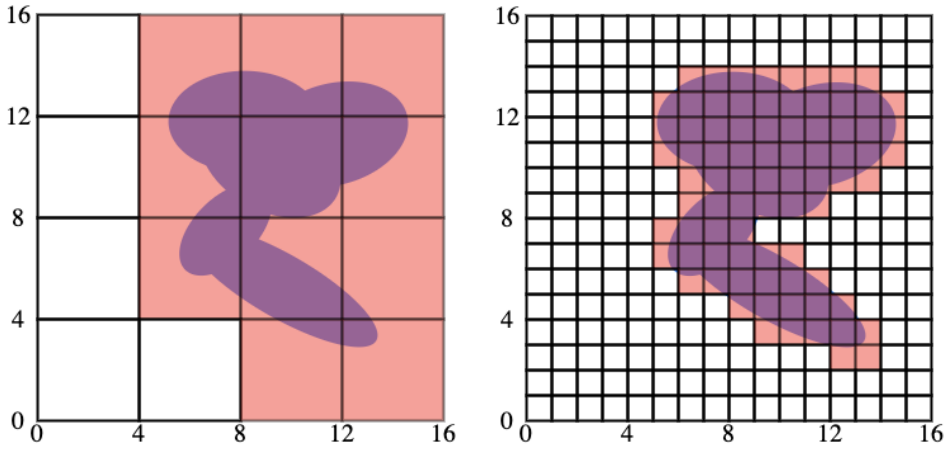


Figure 2.2: **Occupancy Grid Maps for a (16×16) Workspace of Different Resolutions**. Figure **??** shows how an OGM with low resolution, while simpler to construct and analyse, will over-represent the obstacle density of a workspace. Figure **??** shows how a higher resolution will more accurately reflect the obstacles of a workspace.

2.1.2 Rapidly-exploring Random Tree

Rapidly-exploring Random Tree (RRT) is an algorithm designed to efficiently build a tree of collision-free paths in a high-complexity environment. The algorithm grows the tree by randomly sampling points and connecting them to the nearest existing node in the tree. It is inherently biased to grow towards large unsearched areas of the workspace. RRT was developed by S. LaVelle[**?**] and J. Kuffner[**?**]. It is frequently used in autonomous robotic motion planning problems such as autonomous drones.

Scope

RRT takes an initial configuration, a goal point, and an Occupancy Grid Map (OGM) as its input. This OGM may be built and updated using *a priori* knowledge, sensor data from the robot, and other inputs. The algorithm will output a tree of collision free paths toward the goal, as demonstrated in Figure **??**. **It does not calculate the fastest path from that tree**; that can be accomplished using algorithms such as Dijkstra's algorithm.

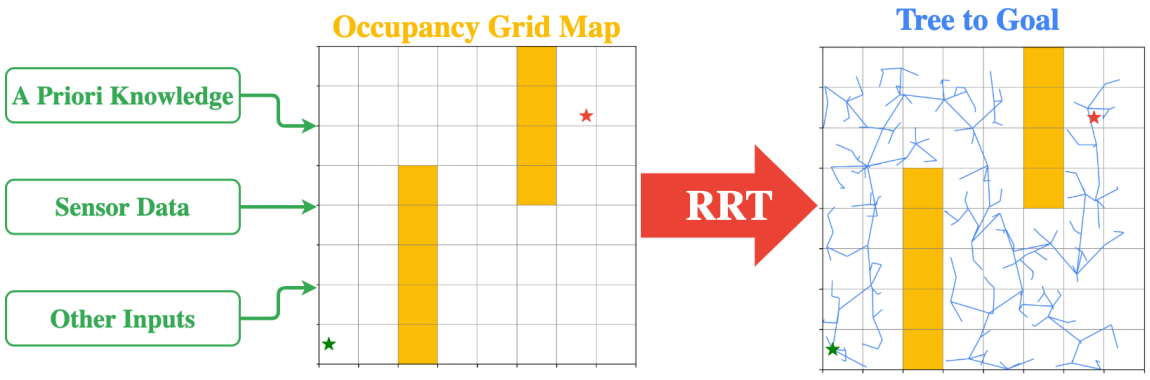


Figure 2.3: **Scope of the RRT Algorithm**: Takes an OGM as input and outputs a tree of collision free paths. The tree is shown in blue on the right.

Algorithm

Put simply, RRT finds a path from start to finish by randomly exploring a workspace. Put more technically, it builds a tree of possible configurations (also known as a graph), connected by edges, for a robot of some physical description. It does so by selecting random configurations and adding them 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 probabilistically complete. The pseudo-code for RRT can be seen in Algorithm **??**

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

Inputs: Initial configuration q_{init},
Number of nodes in graph K,
Incremental Distance ϵ

Output: RRT Graph G with K configurations $[q]$ & edges $[e]$

G.init() **for** $k = 1$ to K **do**
  $q_{rand} \leftarrow$ randomConfiguration()
  $q_{near} \leftarrow$ findNearestConfiguration(q_{rand}, G)
  $q_{new} \leftarrow$ stepFromNearest(q_{near}, q_{rand}, Δq)
  G.addVertex(q_{new})
  G.addEdge(q_{near}, q_{new})
end

Algorithm **??** can be visually represented in Figure **??**, with the example showing a 2D robot operating in a 2D workspace.

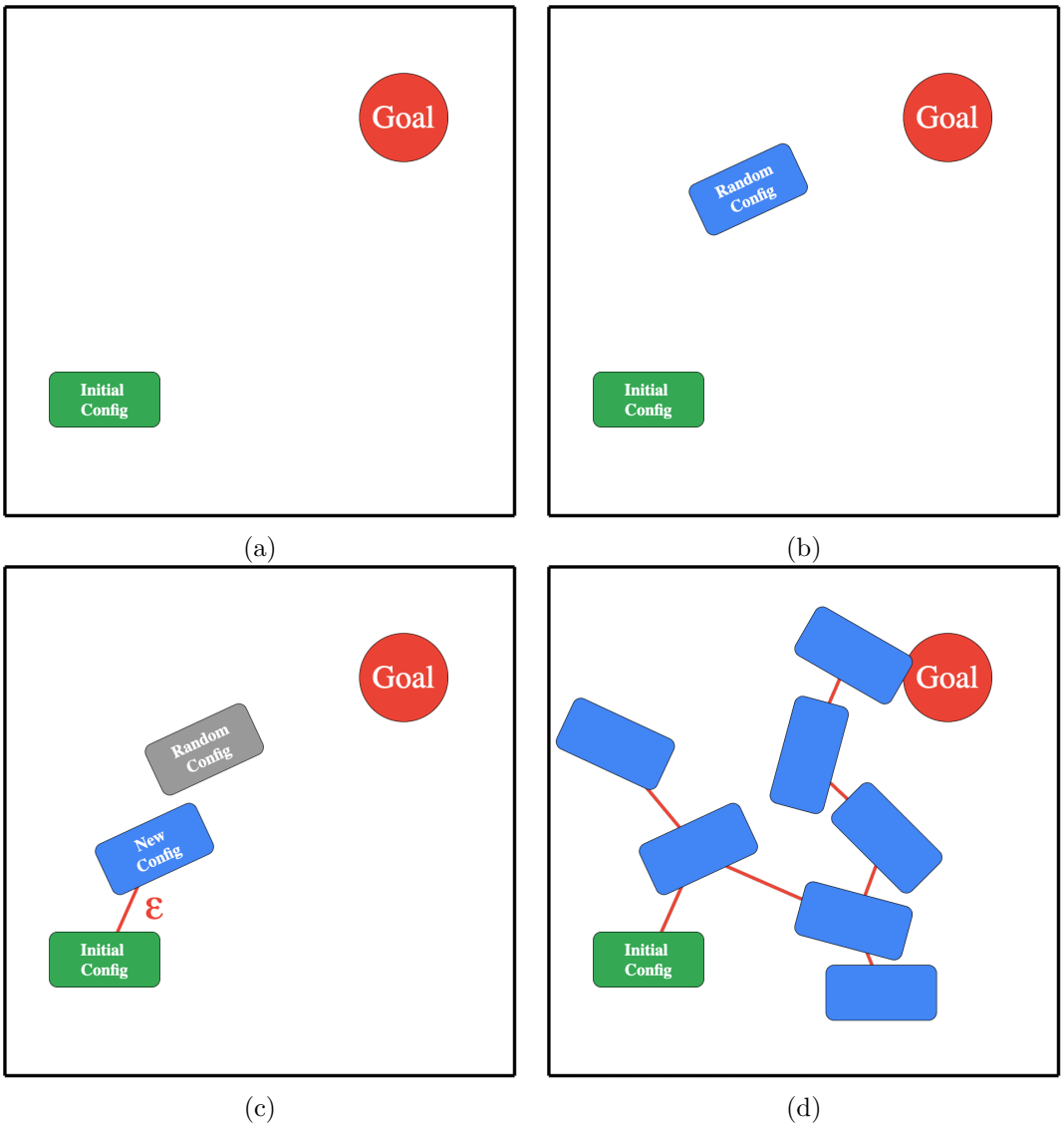


Figure 2.4: **Demonstration of RRT Algorithm for 2D robot in 2D space.** In this example, the graph G begins with an initial configuration and a goal. In (b), the first random configuration is generated. In this case, the nearest configuration is the inital configuration. The random configuration is more than ϵ from the initial configuration, so a new configuration in the direction of the random configuration is generated and added to the graph in (c). This is repeated K times, until the graph in (d) is generated.

Algorithm **??** shows how RRT builds a graph of possible configurations connected

by edges in a completely free configuration space. However, in real-world applications, a robot's workspace space will contain 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).

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

Algorithm 2.2: Rapidly-Exploring Random Tree with Collision Detection

Inputs: Initial configuration q_{init},
Number of nodes in graph K,
Incremental Distance ϵ,
Space S containing obstacles

Output: RRT Graph G with K configurations $[q]$ & edges $[e]$

```
G.init();
for k = 1 to K do
    while !configCollision(q_new) do
        q_rand ← randomConfiguration();
        q_near ← findNearestConfig(q_rand, G);
        q_new ← stepFromNearest(q_near, q_rand, Δq);
    end
    e_new ← 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 Implementation of RRT

2.2.1 Technical Specifications

With RRT selected as the benchmark algorithm against which to test specialized hardware, this project required an implementation of the algorithm that satisfied the following criteria shown in Table **??**. Appendix **??** is a more thorough description of the technical specifications for the implementation of RRT.

Requirement	Brief Description and Justification
Implemented in C/C++	Implementations in C allow for more accurate analysis of computational bottlenecks, unlike higher-level languages like Python.
3D Workspace	The computational requirements of RRT in 3D differ somewhat to that in 2D. Since autonomous UAVs operate in 3D space, it was neccesary to have a 3D implementation to analyse.
UAV modelled as a 3D rectangular prism	In theory, it is possible to model a UAV much more precisely than a rectangular prism. However, in reality, modelling a UAV as a 3D rectangular prism, defined by coordinates $\{x, y, z\}$ and Euler angles $\{\alpha, \beta, \gamma\}$, is more than sufficient (and more computationally efficient). See Appendix **??** for justification.
Mathematically Complete Collision Detection	When RRT is implemented for educational purposes, the edge collision calulations are often simplified to a sampling model which is probabilistically complete but not mathematically complete. In other words, it will catch most collisions by sampling a number of points along each edge, but there is always a possibility of an undetected collision. In real world applications, collisions must be calculated by method of geometric intersection to ensure all collisions are detected.
Highly Parameterizable	Accurate analysis of the algorithm required the ability to vary the following parameters: • ϵ (Maximum distance between two configurations) • K (Maximum number of configurations) • DIM (The upper bound of each dimension for a $DIM \times DIM \times DIM$ workspace) • Goal Bias (How biased RRT is to move towards goal point)

Table 2.1: Abbreviated Technical Specifications for RRT Implementation

The original intention was to find an existing implementation of RRT that could fulfill these requirements. However, no open-source implementations were suitable. Appendix **??** shows an evaluation of existing implementations.

As a result, it was necessary to build a C implementation of RRT from the ground up to the aformentioned specifications.

2.2.2 Implementation Design

The design and implementation of RRT, while neccessary, was significant and time consuming. Since this was not the main object of this thesis, only a brief description of key design choices has been included here. Appendix **??** contains a more detailed account.

System Diagram Here

Parameterization

Table **??** shows the parameters that were included in the implementation and compiled by way of a C header file.

Parameter	Data Type	Description
ϵ	Integer	Maximum distance between two configurations
K	Integer	Maximum number of configurations in the graph
DIM	Integer	Upper bound of each axis of workspace
Goal Bias	Float	Percentage likelihood of stepping towards goal node
OGM	File Pointer	CSV of booleans to represent grids

Table 2.2: RRT Implementation Parameters

Dimensionality

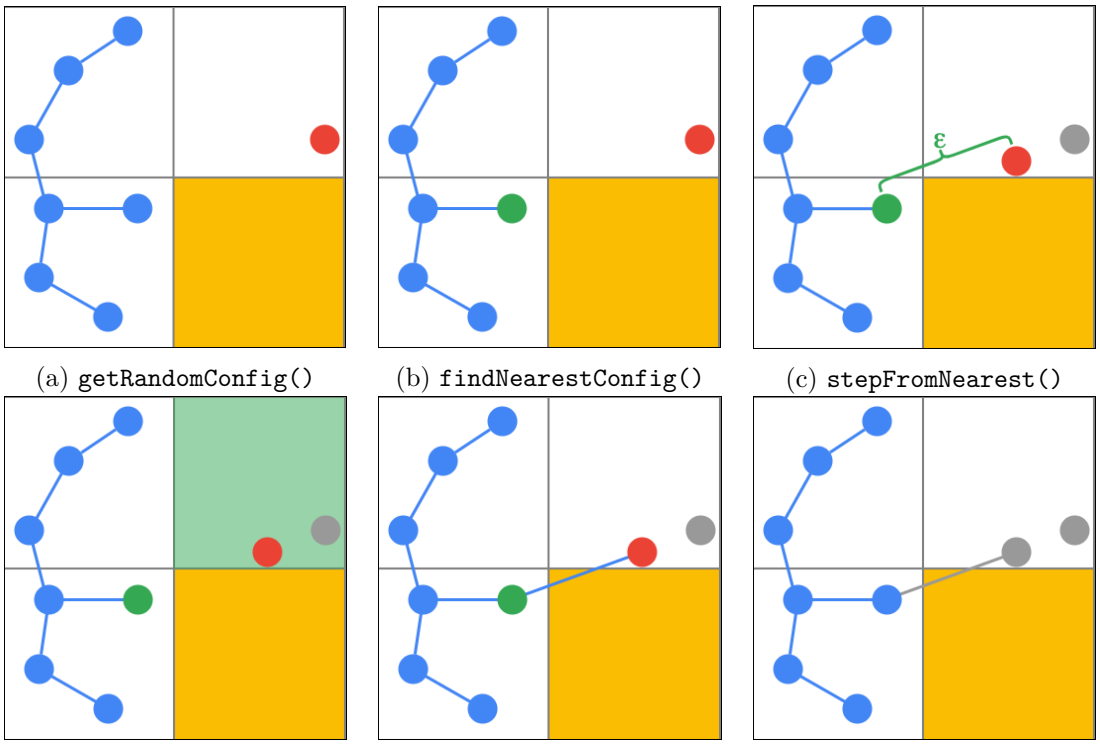
RRT was implemented in both 2D and 3D. Not only did a 2D implementation provide a good development checkpoint, it was also interesting to see the difference in computational load between 2D and 3D, shown in Section **??**.

Modelling a UAV

The UAV was modelled as a 3D rectangular prism, with its configuration represented by Cartesian coordinates (x, y, z) and Euler angles (α, β, γ).

Key Functions

Algorithm **??** shows that there are 5 key functions that constitute RRT. Figure **??** demonstrates each of these functions: `getRandomConfig()`, `findNearestConfig()`, `stepFromNearest()`, `(configCollisions)`, and `edgeCollisions()`. Appendix **??** shows in detail how each of these functions was implemented.



(a) `getRandomConfig()` (b) `findNearestConfig()` (c) `stepFromNearest()`

(d) `configCollision()` (e) `edgeCollision()` (f) Discard

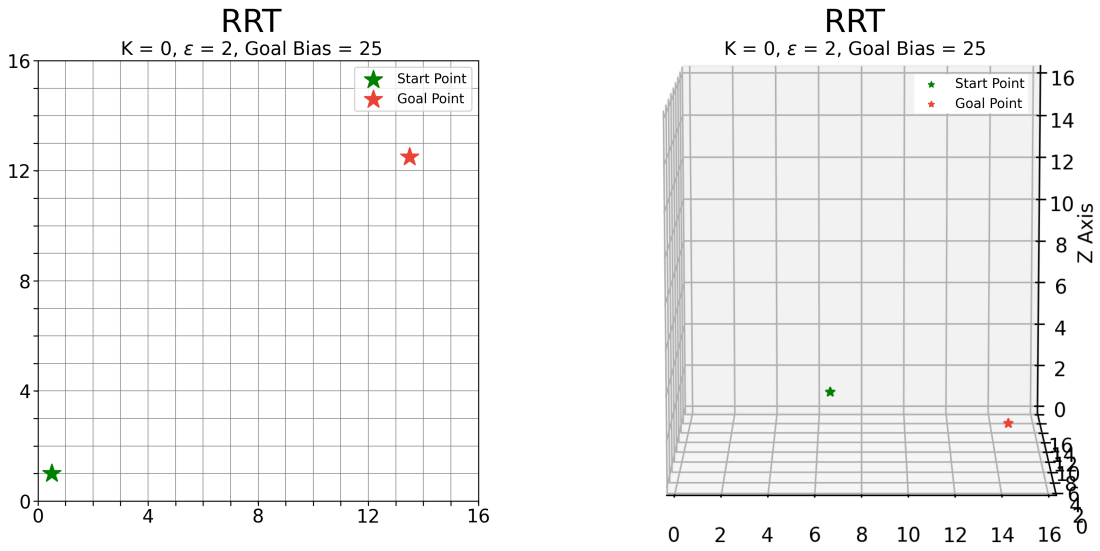
Figure 2.5: **Demonstration of the 5 Key Functions that Constitute RRT**, where configuration is a node in a 2D workspace. A new node is generated in (a) with `getRandomConfig()`, and the closest existing node is found with `findNearestConfig()` in (b). In this case, the new node is further that ϵ from the nearest node, and so a new node is generated with `stepFromNearest()` in (c). `configCollision` determines that the new node is not in an occupied grid (d) and draws an edge between the two nodes. `edgeCollision()` determines that, in this case, there is a collision (e) and the new node is discarded (f).

2.2.3 Implementation Visualization

With the back end functionality of RRT designed and implemented, it was neccesary to develop a way to visualize it. Many existing implementations had the visualization interface run synchonously alongside RRT. This would distort any performance analysis results, and so in this implementation it was left until after RRT had finished executing and then plotted using Python.

Plotting Configurations and the Workspace

Plotting the workspace using the "matplotlib" library was relatively simple in both 2D and 3D, shown in Figure **??**. It was decided that the UAV's' configuration would be visualized only as its origin point, rather than plotting a 3D rectangular prism at each configuration, in order to maintain simplicity. Nevertheless, the UAV was still modelled as a 3D prism in the backend.



(a) Workspace in 2D

(b) Workspace in 3D

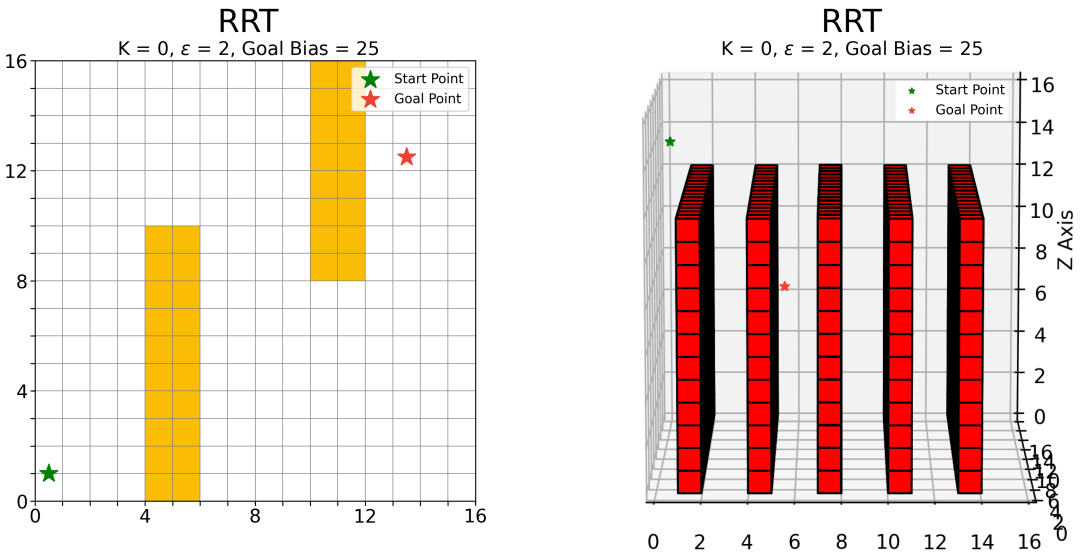
Figure 2.6: **Visualization of Workspace in 2D and 3D**, with configuration represented by only a point, and Start and Goal nodes shown

Plotting Obstacles

Obstacles were plotted in accordance to the input OGM, shown in Figure **??**

Plotting RRT Graph

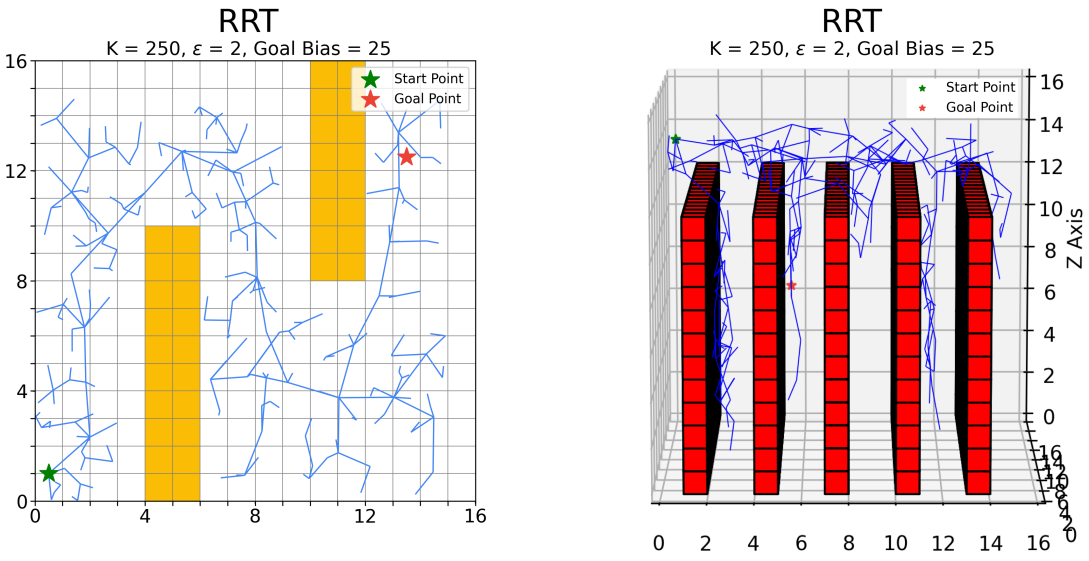
To keep the plot simple, it was decided to not show the origin point of each configuration in the graph produced by RRT. Instead, only the edges of the graph were plotted, seen in Figure **??**



(a) Obstacles in 2D

(b) Obstacles in 3D

Figure 2.7: **Visualization of Obstacles in 2D and 3D**, Obstacles shown in yellow and red for 2D and 3D respectively.



(a) RRT in 2D

(b) RRT in 3D

Figure 2.8: **Visualization of Obstacles in 2D and 3D**, with Graph shown in blue.

2.3 Analysis of RRT

Having implemented a functioning version of RRT that adhered to the specifications set out in Table **??**, analysis of its computational profile could begin. The purpose of this analysis was to identify the biggest bottleneck of RRT and therefore the best opportunity for hardware acceleration.

2.3.1 Experimental Methodology

Experiments were set up to determine which of the 5 key functions of RRT takes up the biggest share of computational load. The only fair way of determining the computational load of each function was to measure the percentage of CPU time each function takes for the **fastest possible execution** of RRT for a given map size. This is explained in more detail in Section **??**.

Measuring Performance

The "performance" metric of interest is the percentage of total time the CPU spends executing each of the 5 key functions. CPU analysis of a program can often be more complicated than merely timing how long each function takes to execute. Software can be written with inbuilt multithreading and other optimizations that require special CPU analysis software, such as Intel's VTune Profiler[**?**]. This software is designed to find computational bottlenecks in large, complex programs. However, it takes significantly longer to run (which was unsuitable for running hundreds of thousands of tests), and is less customizable, than adding performance timers directly to the program's code. It was also hypothesized that, since this project's implementation of RRT did not use multithreading or any other timing distorting optimizations, custom performance tracking should yield the same results as VTune Profiler. As such, custom performance tracking was added to the RRT implementation. This custom performance tracking method was verified by conducting a χ^2 test against data from VTune Profiler, and was found to be accurate. Appendix **??** gives more detail on timing methodology.

Optimal Parameters

Extensive testing was undertaken to determine the optimal parameters for a given map size. The goal was to find the set of parameter values for which RRT would reach its goal with $\geq 98\%$ probability, for a wide variety of OGMs, in the shortest possible time.

For each map size $\{4, 8, 16, 32, 64\}$,the parameters that were varied were ϵ, K, and Goal Bias. The success rate and average execution time was measured by, for each set of parameter values, running RRT 100 times. Thus, with 5 different map sizes, if 4 values were tested for each parameter, and 4 different OGMs were tested, the total number of tests $= 4^4 \times 5 = 1280$ (with each test running RRT 100 times!)

As such, only the optimal parameter values for each map size are in included in Table **??**.

DIM	K	ϵ	Goal Bias (%)	Success Rate (%)
2D				
4×4	75	1	10	100
8×8	100	2	25	98
16×16	125	4	25	99
32×32	250	8	10	100
64×64	500	16	25	100
3D				
$4 \times 4 \times 4$	75	1	10	99
$8 \times 8 \times 8$	100	2	25	100
$16 \times 16 \times 16$	100	4	25	100
$32 \times 32 \times 32$	250	8	10	99
$64 \times 64 \times 64$	500	16	25	100

Table 2.3: **Optimal RRT Parameters for each Map Size**, shows the optimal set of parameters after extensive testing, alongside their respective success rates over 100 executions of RRT for different OGMs.

2.3.2 Results

As expected, the total execution time of RRT for optimal parameters increased with the size of the map, shown in Figure **??**.

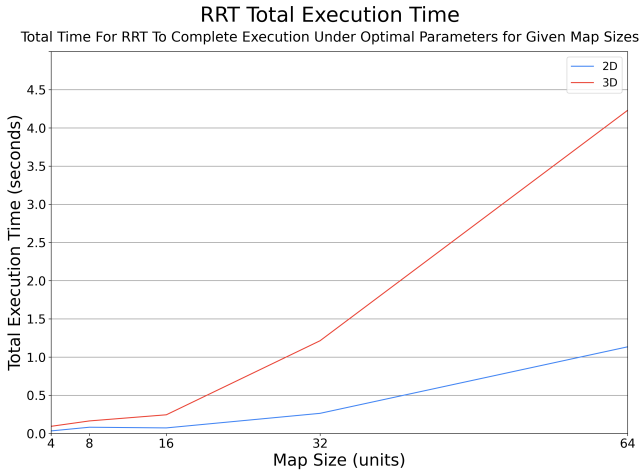


Figure 2.9: **Increasing Total Execution Time of RRT with Map Size**

2D Computational Load Profile

Figure **??** shows that the two biggest computational loads are `findNearestConfig()` and `edgeCollisions()`, with the latter increasing as the size of the map increases. The fact that the load of `edgeCollisions()` takes the majority of execution in bigger map sizes means that, at least in 2D, it can be considered the bottleneck function.

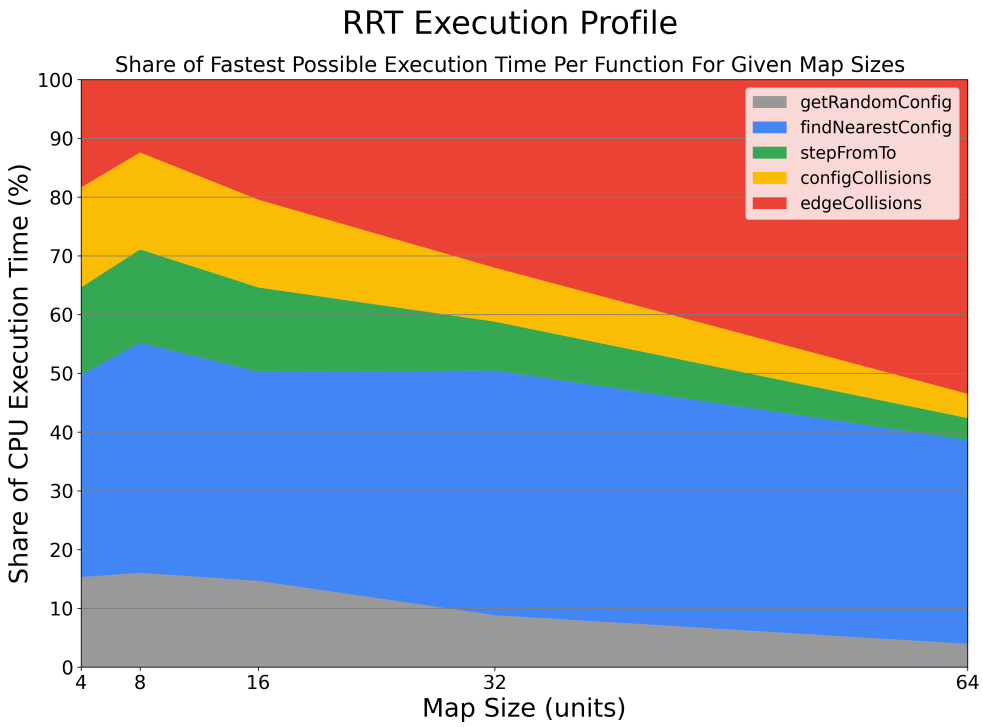


Figure 2.10: **Profile of Computational Load of RRT in 2D**

3D Computational Load Profile

The computational load of `edgeCollisions()` was even greater in 3D, starting at 40% for $4 \times 4 \times 4$ maps and increasing to 70% for $64 \times 64 \times 64$ maps, as shown in Figure **??**.

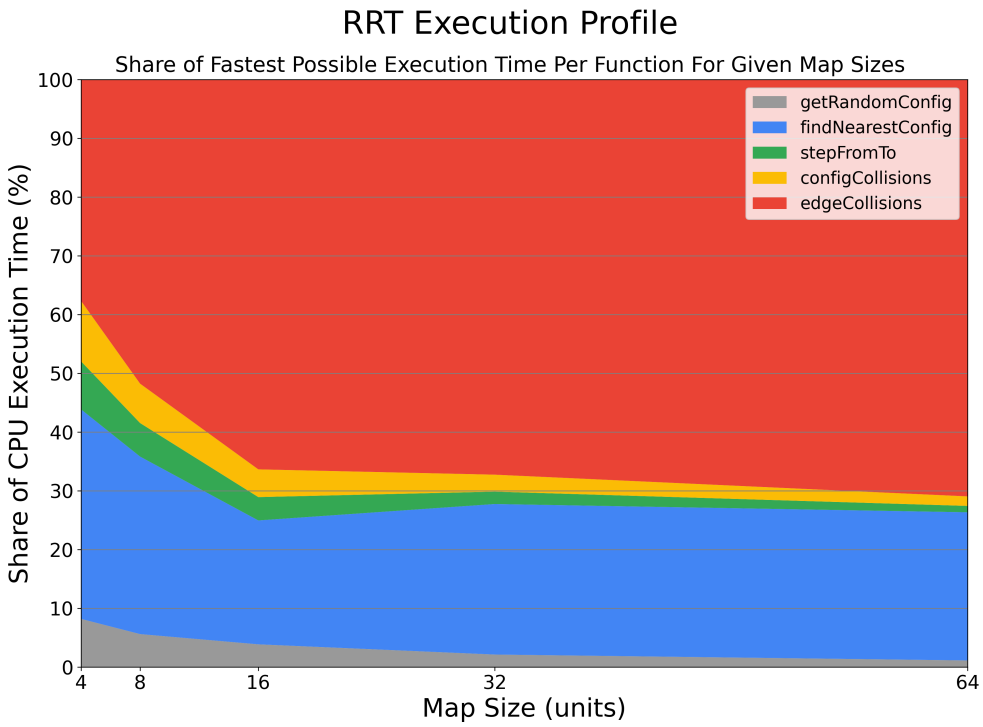


Figure 2.11: **Profile of Computational Load of RRT in 3D**

As such, it is safe to say that the bottleneck function for RRT is `edgeCollisions()`. This conclusion is strengthened by the fact that `edgeCollisions()` was implemented in the fastest possible way (without relying on approximations or implementing multithreading), whereas `findNearestConfig()` was implemented without any optimizations (a possible optimization was the K-nearest node algorithm, for instance). Finally, this conclusion supports prior research that collision detection takes up the vast majority of CPU execution time. As such, this is the function that was targeted for hardware acceleration.

Chapter 3

Motion Planning in Hardware

The second objective of this thesis was to design and implement a functional hardware unit that accelerates the execution of RRT in 3D. With the bottleneck function having been identified in Chapter 2 as edge collision detection, Chapter 3 details the specification, design, implementation, and analysis of a hardware unit the implements the edge collision function.

3.1 Defining the Collision Detection Unit

3.1.1 Edge Collision Function

To briefly examine the edge collision detection function in general terms; Given an edge e, RRT finds where e intersects with grids in the OGM. If any of the grids it intersects with are "occupied", a collision is returned. This is shown in Figure **??** on Page **??**.

To see why it is so computationally intense to calculate intersections between a segment and grids, it must be understood that it is a fairly involved geometric process. Figure **??** on Page **??** shows how grid intersections are detected by computing the where the segment intersects certain **axis-oriented planes**.

Time Complexity

With the steps of the edge collision algorithm understood, (explained graphically in Figure **??**, algorithm included in Appendix **??**, its Time complexity may be quantified. For an edge e of maximum length ϵ, it must check for intersections with $\epsilon \times \epsilon \times \epsilon$ grids. (i.e the only grids that are checked are the ones that the e could possibly intersect with). It first iterates through the three dimensions of axis-oriented planes (xy, xz, and yz). This is a constant of 3. Within each of these dimensions, it must iterate through ϵ planes. This makes its time complexity $O(3\epsilon)$.

Appendix Reference?

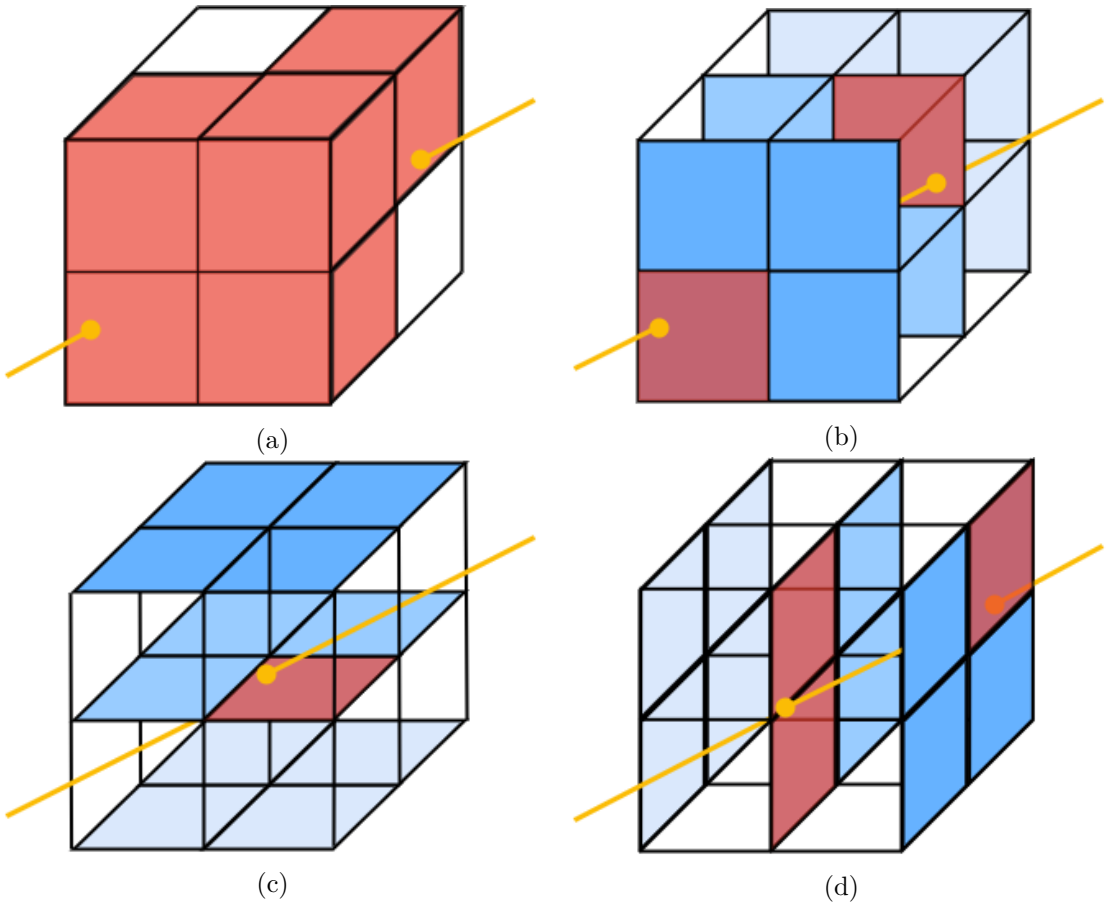


Figure 3.1: **Detecting Grid Intersections by Finding Intersections with Axis Oriented Planes**

Consider an edge e that spans the width of a $2 \times 2 \times 2$ grid map, as shown in Figure **??** (do not consider if the grids are occupied, this is just to determine which of them the edge intersects). Just by eyeballing Figure **??** it seems odd that so many of the grids have been intersected (denoted by red shading) by the yellow edge. The algorithm executed by checking one set of axis-oriented planes at a time. Figure **??** shows how the xy-oriented planes are checked for 3 different values of z (going into the page), finding two intersection points. There is only one intersection for the xz oriented planes (**??**). In Figure **??**, the segment intersects 2 grids in the second yz-oriented plane, and one in the third plane. The intersected grids are thus any grid where a point-of-intersection falls on its face.

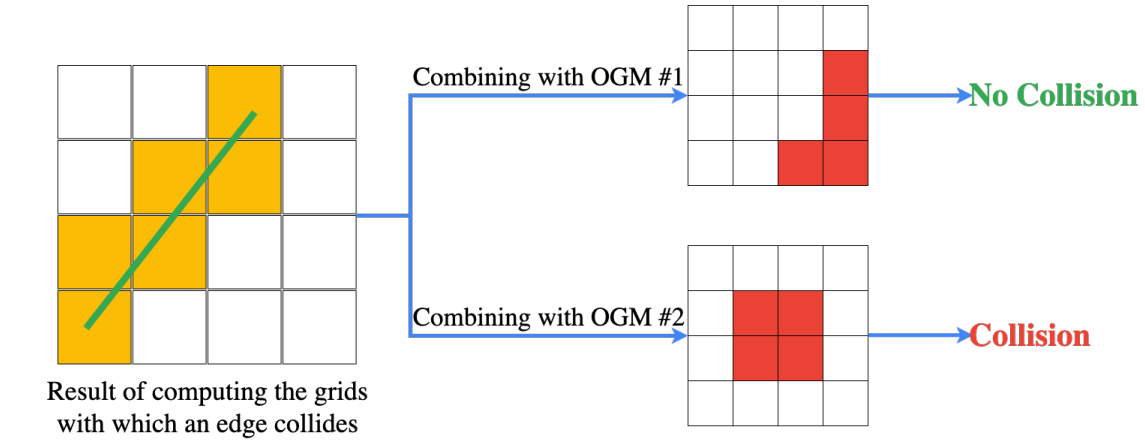


Figure 3.2: **Edge Collision Computation Process**. Most of the computational load is found in determining the grids with which an edge intersects (as seen on the left of the figure). Once these grids have been found, it is very simple (and fast) to lookup if these grids are occupied in the OGM

3.1.2 Technical Specifications

Performance Specifications

When accelerating motion planning algorithms, it is often difficult to quantify a goal for how much faster one would like the function to run - the answer is usually "as fast as possible!" For this thesis, the performance specification was set to be the edge collision function to run fast enough such that it was no longer the bottleneck function. This translated to a desired speedup of about 3 times (when compared to benchmark performance of a typical CPU). Table **??** quantifies this in terms of latency and throughput.

Metric	Benchmark CPU*	Accelerated
Latency (μseconds/edge)	2.6	0.9
Throughput (edges/second)	384,615	1,111,111

Table 3.1: **Performance Specifications for Edge Collision Detection Unit**. *Benchmark CPU is an Intel 3.1 GHz i7 Dual Core processor, typical of a laptop computer.

Area Specifications

Generally, an inverse relationship exists between latency and area. While it may be possible to make the unit much faster than the latency specification, this may become prohibitive with regards to the amount of area on chip it would occupy. It was decided to limit the area to that which would fit on an FPGA typical in drone applications (those of the Kintex-7 Low Voltage family were chosen, but there are many possible options.)

Logic area on an FPGA is largely determined by Look-Up Tables(LUTs). This is a "truth-table" used in FPGAs that determine what output to return for a given input. Any amount of combinational logic can be reduced to a number of truth tables. As such, the upper bound on area was set at 274,080 LUTs.

Interface Specifications

As shown in Figure **??**, the computationally intensive part of the process of edge collision detection is finding points of intersection between an edge and the grids of the map. Comparing this result to an OGM is simple and fast. Therefore, it was decided that the hardware unit would simply take an edge and determine the grids with which it intersects. Whether the edge intersects a given grid can be represented as a binary $\{0, 1\}$, and thus a the intersections found in a $\epsilon \times \epsilon \times \epsilon$ gridspace can be represented as an ϵ^3 sequence of binary values. Table **??** outlines the required interface specifications for the functional unit.

Element	**Description/Justification**
Constraints	
Length ϵ	ϵ defines the max edge length. The space being checked and the output sequence has the dimensions $\epsilon \times \epsilon \times \epsilon$
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 $\{x, y, z\}$.
Control Inputs	The functional unit must have ports for control signals: clock, reset, start. These are required for adding the unit to a processor.
Outputs	
Return Value	ϵ^3 bit sequence: 1 if collides with grid at that index, 0 otherwise.
Control Outputs	Output ports for control signals: idle, done, ready. These are required for adding the unit to a processor.

Table 3.2: **Interface Specifications for Edge Collision Detection Unit**

3.2 HoneyBee

Figure 3.3: **Megalong Park Honey Bee Pollinating a Weeping Cherry Blossom**. Photographed by Emma Kenny in the Southern Highlands of New South Wales, Australia

The honey bee, *Apis mellifera*, has long been renowned for its tireless work ethic. However, the it is rarely given credit for its remarkable navigation and collision avoidance strategies during flight. Recent research[**?**] suggests that honey bees, interestingly enough, explore their workspace randomly in order to find paths from their hive to sources of pollen. Sound familiar? As such, it is quite appropriate that this functional unit, designed to work tirelessly, rapidly and efficiently to execute collision detection computations for robot motion planning, was named **HoneyBee**.

HoneyBee is a hardware unit that will eventually be incorporated into a processor, demonstrated in Figure **??**. In Chapter 4, the HoneyBee unit is implemented in a simple RISC-V processor and invoked using custom RISC-V instructions. For now, however, consider HoneyBee as a standalone unit that computes the grids with which an edge collides. Its resulting output can be compared to an OGM, as explained in section **??**.

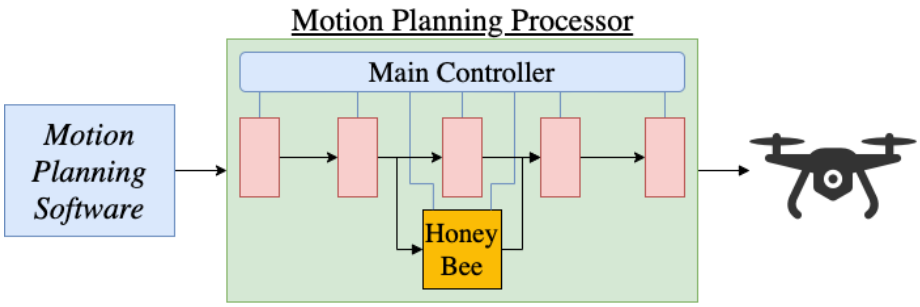


Figure 3.4: HoneyBee in a Motion Planning Processor
. Shown is an abstraction of how HoneyBee would become an extension of the normal processor datapath.

3.2.1 HoneyBee Interface Design

The interface for the HoneyBee functional unit, following on from the interface specifications outlines in Section **??** can be simply represented by Figure **??**.

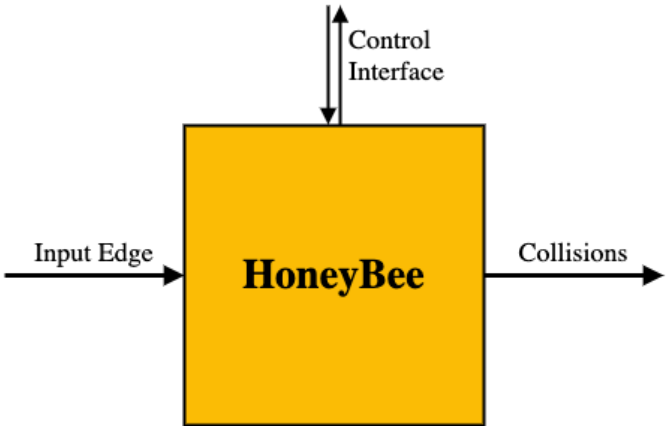


Figure 3.5: **General Overview of HoneyBee Interface**. The functional unit takes an edge e, defined by two points p_1 and p_2, as an input, and outputs a series of collisions. These collisions describe which grids an edge intersects. Its control interface allows for communication with a processors main control unit.

However, when designing hardware (the method of doing so is described in section **??**), how these inputs and outputs are implemented must be considered at the bit level. Figure **??** shows all input, output, and control ports, and their bit-widths.

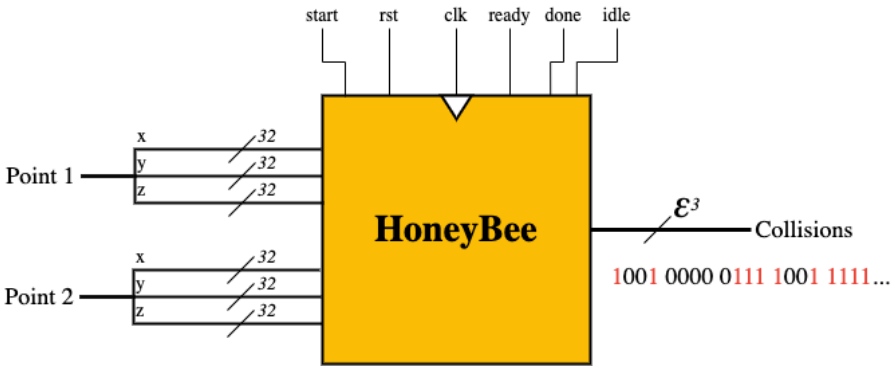


Figure 3.6: **Port Diagram of HoneyBee Interface**. The edge input is represented by 6 32-bit floats, following the IEEE 754 Single Precision 32-bit protocol, each float representing one of its coordinate points. The output sequence of collisions is a e^3-bit sequence, with each bit in the sequence representing one of the grids that was checked for intersections. It has input control signals for start and reset, and output control signals for done, idle, and ready. These control signals make up the necessary signals for a handshake protocol between HoneyBee and a processor's main controller.

Inputs

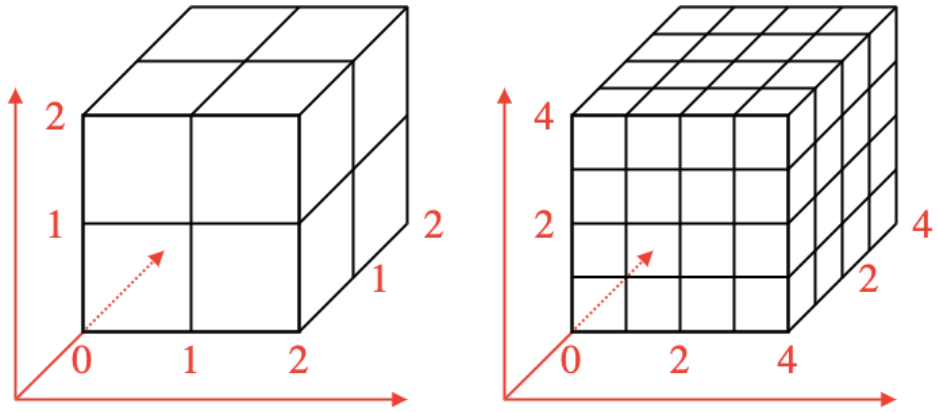
The inputs to HoneyBee collectively describe a single edge. This is done with 6 32-bit floating point numbers. How floating points (non-integer numbers) are represented in binary determined by the IEEE Standard for Floating-Point Arithmetic (IEEE754). How this is actually represented is not neccesary to understand, but is explained in Appendix **??**. The important point is that the input edge determined by 6 32-bit coordinate points.

Output

HoneyBee outputs a sequence of "collision-bits," with each bit in the sequence representing if the input edge collides with its corresponding bit. How this sequence of bits is mapped to a 3D grid-map is explained in Appendix **??**. It is important to note that in the design and implementation of HoneyBee, the length of this sequence was parameterized to be variable, corresponding to a variable value of ϵ. Recall that the optimal edge collision algorithm only checked ϵ^3 grids. HoneyBee, as well, only checks the grids with which the edge could possible intersect.

Since the number of grids being checked is parameterized, so must the number of collision-bits be. This is demonstrated in Figure **??**.

Note: The output bit-width is *parameterized* not *variable*. Upon synthesis (building) of HoneyBee, the output bit-width is set at a constant value. Different syntheses may have different output bit-widths. When the time comes to add HoneyBee to a processor, it is



(a) For $\epsilon = 2$, an output collision-bit sequence of length 8 is required.

(b) For $\epsilon = 4$, an output collision-bit sequence of length 64 is required.

Figure 3.7: **The Impact of ϵ on the Length of the Bit-Collision Sequence**

synthesized with a certain bit-width.

Control Interface

The control interface is designed to give HoneyBee the ability to be included in a processor, and implements a commonly used "handshake" protocol between HoneyBee and the control unit of the processor in which it resides. Put simply, this is a method that allows the control unit to tell HoneyBee when to start executing the computation, and for HoneyBee to tell the control unit when it has finished its computation and the output value is ready. This is explained in detail in Appendix **??**. The control interface also has a clock and reset port.

3.2.2 HoneyBee Implementation

Hardware Description Languages

Designing computers and their constituent parts has come a long way from its arduous beginnings. "Victory", the enigma-breaking machine designed by Alan Turing at Bletchley Park during World War II, was a large electro-mechanical computer made up of storage wheels, electromagnetic relays, and rotary switches, assembled by hand.[**?**] So too was "Mark I", the 816 cubic feet computer designed by Harvard University's Dr. Howard Aiken, which, on March 1944, computed the viability of implosion for detonating the atomic bomb.[**?**]

Computers nowadays measure in the order of millimeters rather than meters. What's more, they are now "built" in software, using a Hardware Description Language (HDL). HDLs is a family of computer programming languages that are used to specify the func-

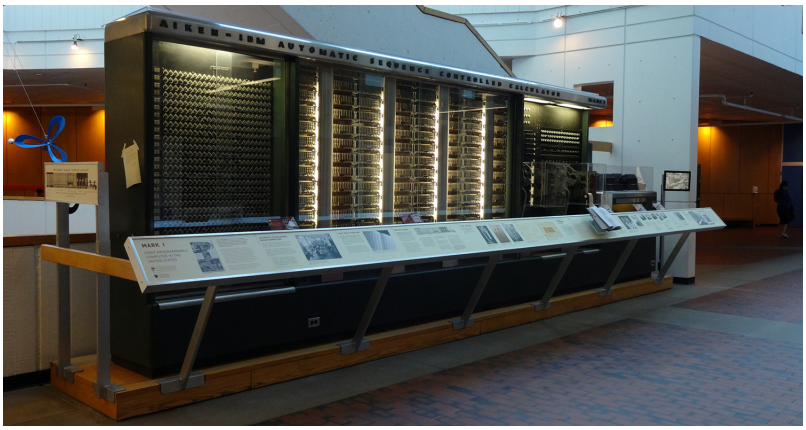


Figure 3.8: **The Harvard Mark I Computer**, photographed in the Harvard Science Center in April 2014 (Source: Harvard University)

tion of electronic circuits. Tools allow for simulation of such circuits to verify design correctness and performance. Modules defined in HDLs may then be synthesized for a type of integrated circuit called a Field Programmable Gate Array (FPGA). This FPGA, "programmed" in HDL code to behave in a certain way, can then serve the purpose of a processor or other functional processing unit. Figure **??** demonstrates this process.

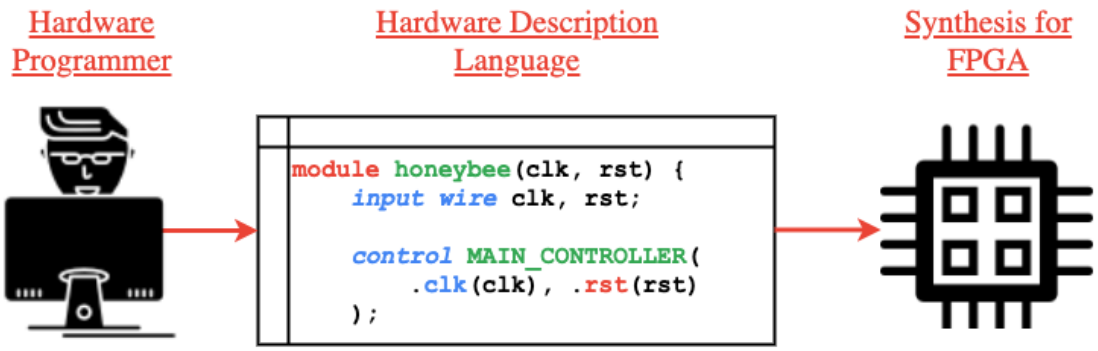


Figure 3.9: **The Hardware Development Process**, Defining Hardware Units in Hardware Description Languages for FPGAs.

HoneyBee was implemented eventually in an HDL called Verilog. However, no Verilog for HoneyBee was ever explicitly written by a human. It was generated by a tool called High-Level Synthesis.

High Level Synthesis

High Level Synthesis (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. In short, it effectively translates programming languages into hardware description languages. Some key advantages of using HLS are 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.

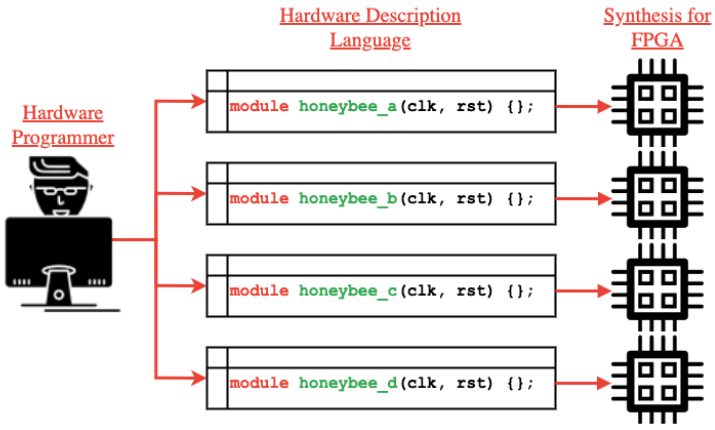
The most important benefit of using HLS, however, is the ability to use "pragmas." These are simply directives given to the HLS tool that tell it what optimizations to use when translating C code into an HDL. This allows the same funtionality to be synthesized in many different ways, optimizing the synthesis for speed, area, memory, etc. As such, the hardware development process now allows developers to experiment quickly with different ways to implement the same functionality. This is demonstrated graphically by Figure **??** on Page **??**.

HoneyBee-A Synthesis

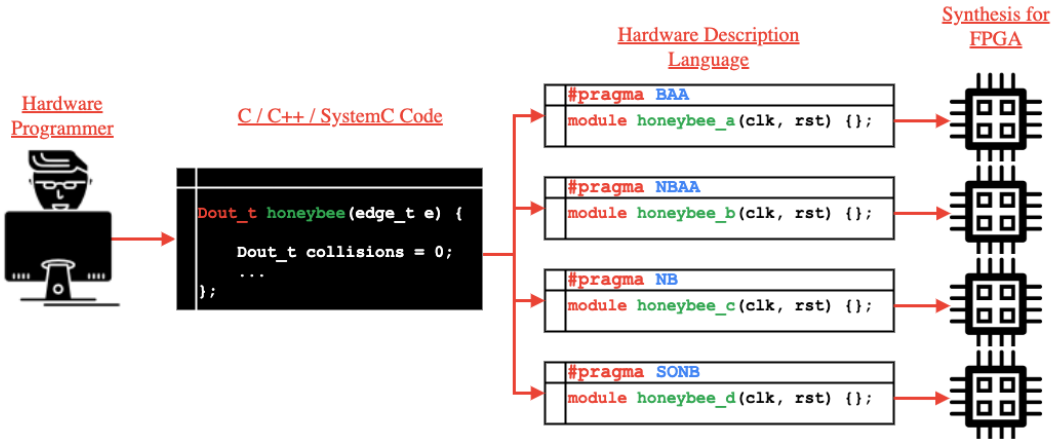
With the functionality of HoneyBee implemented in C, the first optimization iteration (designated HoneyBee-A (HB-A)) was synthesized. HB-A had no pragmas, and was merely a basic hardware implementation of the defined functionality. HB-A and all subsequent iterations) synthesized correctly to the interface specifications (See Appendix **??** for technical interface report of synthesis). Table **??** shows the result of HB-A's synthesis compared with its performance and area specifications. Obviously, the synthesis is well below the area limit, but nowhere near the specifed performance metrics. This is where the beauty of High-Level Synthesis optimization came in.

Metric	**Specification**	**Synthesis Result**
Latency (μseconds/edge)	0.9	6.66
Throughput (edges/second)	1,111,111	150,150
FPGA Area LUTs	274,080	10,593

Table 3.3: **Synthesis Results for HB-A with** $\epsilon = 4$



(a) Hardware Optimization without HLS



(b) Hardware Optimization with HLS

Figure 3.10: **Hardware Optimization Process**. Figure **??** shows how, without using HLS, the hardware developer must write multiple different implementation of the same functionality to acheive different performance. Figure **??** shows how, when using HLS, the hardware developer only must write one implementation (in a higher-level language like C), and then use "pragmas" to create different implementations (with different optimizations) for synthesis.

3.2.3 HoneyBee Acceleration

This section steps through the process of using HLS pragmas in 2 major optimization iterations, HoneyBee-B (HB-B) and HoneyBee-C (HB-C) and how these iterations compared to benchmark and specified performance.

Benchmarking

The benchmark performance was based on a Dual-Core Intel 3.1GHz i7 processor. In an ideal world, this processor would have been chosen after a rigourous process of determining the most suitable benchmark. In reality, this processor was chosen because it was the one found in the computer running the simulations (an early 2015 MacBook Pro). Nevertheless, it serves as a suitable benchmark, demonstrating performance typical of general purpose CPUs.

Examining latency, the benchmark was set at the average execution time for the benchmark CPU to compute 1 edge. Tests were run and averaged over 1000 trials. The average latency was 2.6 μseconds, with a standard deviation of 0.1 μseconds. This is shown, along with the performance of HB-A in Figure **??**.

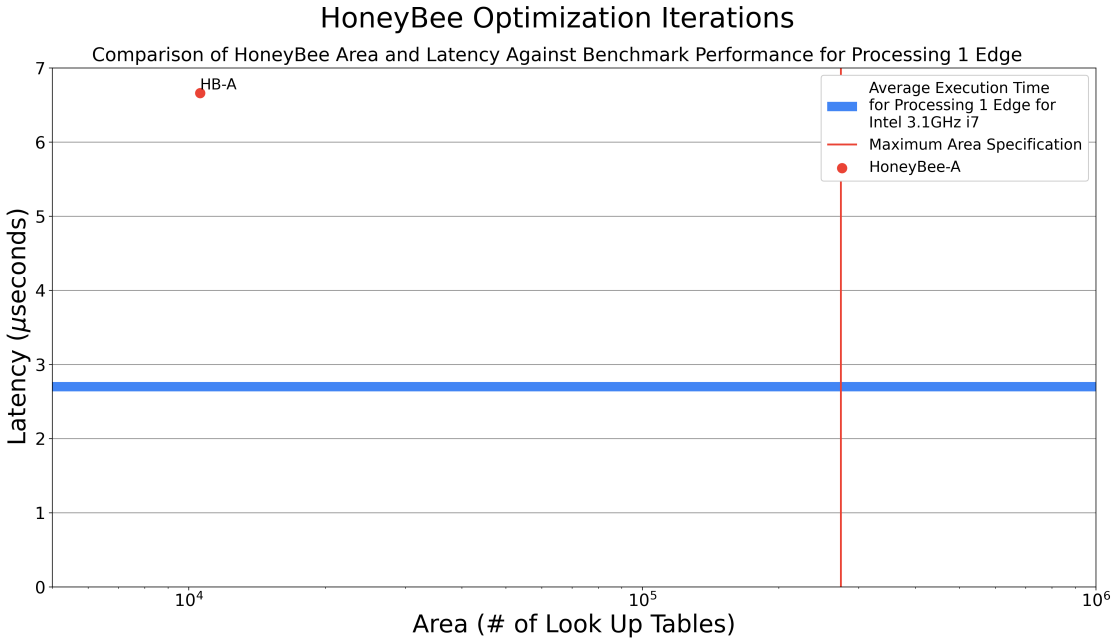
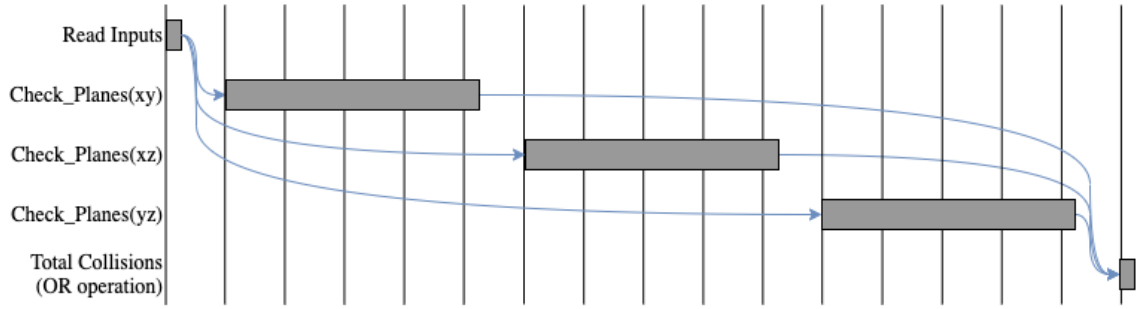


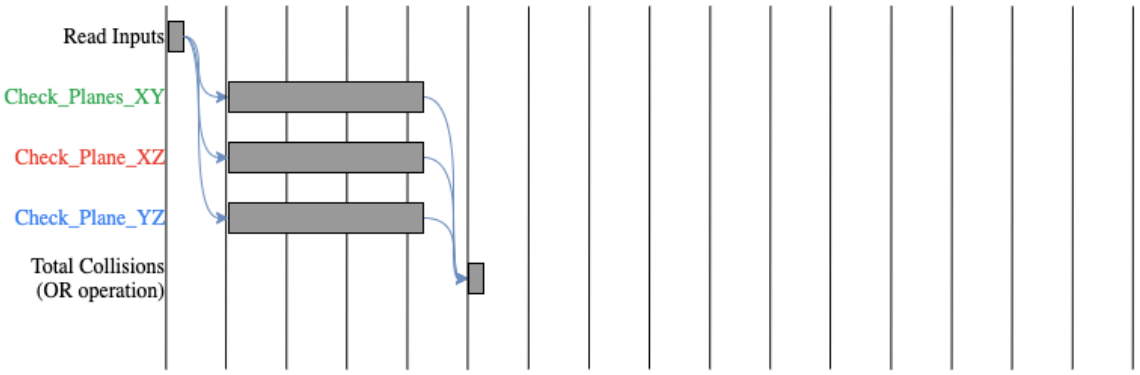
Figure 3.11: **HB-A Performance Against Benchmark CPU**. Standard deviation of CPU average performance is shown by width of the line for easier interpretation.

HoneyBee-B

The first step in accelerating HoneyBee was taking advantage of the inherent parrallelism available to the algorithm. Recall that the edge collision algorithm checks the xy-oriented planes, the xz-oriented planes, and finally the yz-oriented planes. These operations are completely independent, and can thus be performed simultaneously. Figure **??** shows how the process of checking each set of planes was done sequentially in HB-A, but are executed in parallel in HB-B.



(a) HoneyBee-A Timing Diagram. `Check_Planes` executed sequentially.



(b) HoneyBee-B Timing Diagram. `Check_Planes` executed in parallel. Different colors are to represent slightly different implementations of the same function.

Figure 3.12: **Timing Diagrams Showing Parallelization in HoneyBee-B**. Note, these are simlified timing diagrams for easy explanation of the concept of hardware parallelization.

The timing diagram does not only show the use of parallelism. It also shows how this is actually implemented in hardware. In HB-A (Figure **??**), there is a single module named `Check_Plane`. Since there is only one of them, it must calculate intersections with each set of planes sequentially. On the other hand, in HB-B (Figure **??**), there are three seperate instances of this module (`Check_Planes_XY`, `Check_Planes_XZ`, and `Check_Planes_YZ`),

allowing HoneyBee to execute computation on all three sets of planes in parallel.

Moreover, notice that in HB-B, execution time of each of these three instances is shorter than that of the single instance in HB-A. When a single module instance is used for different purposes (in this case, checking the xy, xz, and yz oriented planes), it has some variability. To control this variability, it must execute a certain amount of control logic at the beginning of the function. On the other hand, when there are seperate instances of the module, what was once variable can now be made constant. As a result, each instance can be slightly specialized and the control logic eliminated. In this case, a general `Check_Planes` module was replaced with the three specialized `Check_Planes_XY`, `Check_Planes_XZ`, and `Check_Planes_YZ`. Each of these has less variability, thus less control logic, and therefore a slightly faster overall latency. Figure **??** shows how theoretically, this should result in a reduction in overall latency of more than 3 times.

Comparing HoneyBee-B against our benchmark, the success of this optimization can be seen. Figure **??** shows the performance of multiple variants of HB-B in yellow. These variants were the result of experimenting with slightly different pragmas, but all fell in roughly the same area. Appendix **??** lists the details of each HB-B variant.
HB-B3 showed the best performance/area relationship. It was of acceptable area and had latency marginally lower than the benchmark, but was not as fast as the defined specifications. Table **??** shows the result of HB-B3's synthesis compared with its performance and area specifications.

Metric	**Specification**	**Synthesis Result**
Latency (μseconds/edge)	0.9	2.05
Throughput (edges/second)	1,111,111	487,805
FPGA Area LUTs	274,080	26,524

Table 3.4: **Synthesis Results for HB-B3 with** $\epsilon = 4$

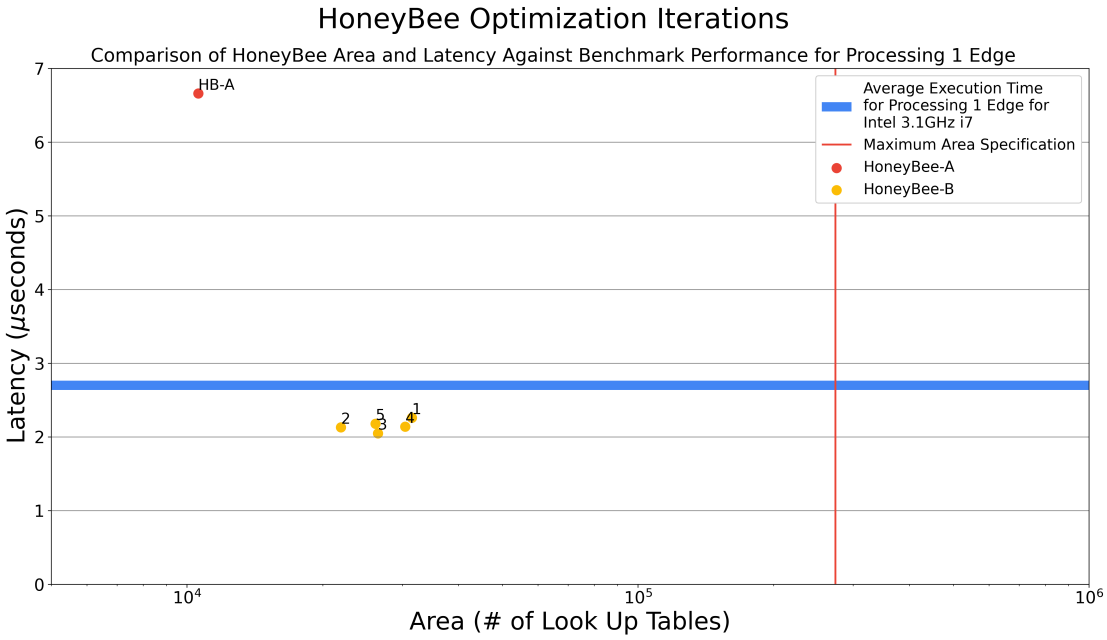
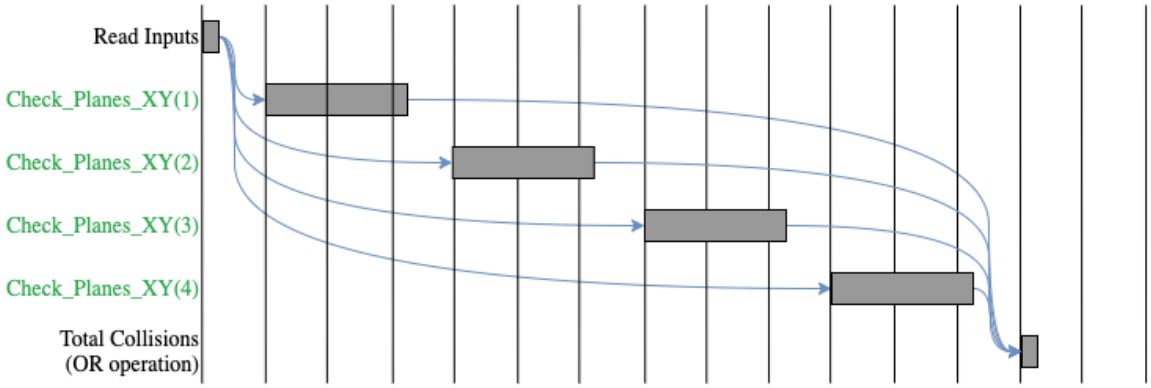


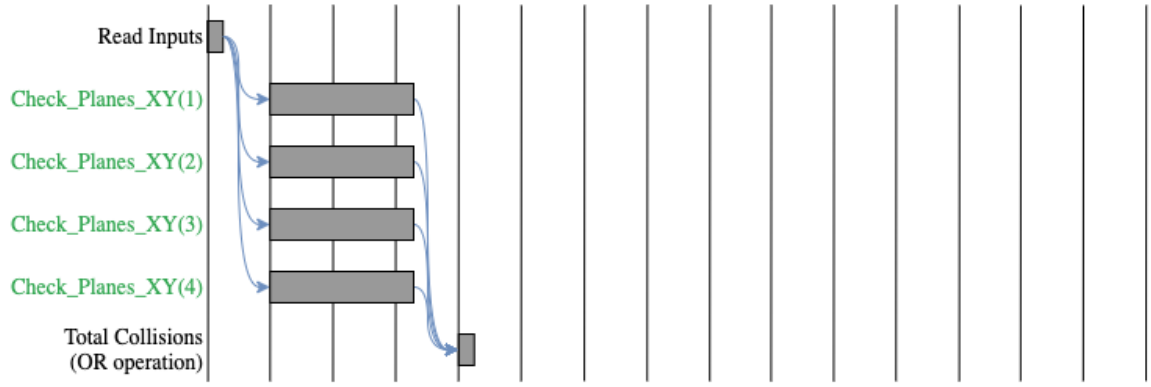
Figure 3.13: **HB-B Performance Against Benchmark CPU**
Variants of HB-B shown in Yellow.

HoneyBee-C

A similar concept was applied to optimize the computation of each set of planes. Consider synthesis of HoneyBee for $\epsilon = 4$. HoneyBee will need to compute intersections with 4 xy-oriented planes, 4 xz-oriented planes, and 4 yz-oriented planes. HB-B computes each set of 4 planes simultaneously, but the computing intersections with each of the 4 planes in one orientation are also independent operations. As such, they can also be parallelized with the instantiation of more hardware modules. Figure **??** shows the timing diagrams for the `Check_Planes_XY` module that was shown in the last set of timing diagrams.



(a) HoneyBee-B Timing Diagram for `Check_Planes_XY`. One instance of `Check_Planes_XY` module executed sequentially 4 times ($\epsilon = 4$.



(b) HoneyBee-C Timing Diagram. 4 instances of `Check_Planes_XY` module executing in parallel.

Figure 3.14: **Timing Diagrams Showing Parallelization in HoneyBee-C**. Again, these are simplified versions of timing analysis for easy explanation of the concept of hardware parallelization.

Just focussing on the computation of the xy-oriented planes, Figure **??** shows how HB-B than having one instance of the module execute 4 times sequentially. HB-C implements greater parallelism by instantiating 4 module instances to execute sequentially. HB-C, as a result, exceeded the performance benchmark and was of an acceptable area, as shown in Table **??** and Figure **??**

Metric	Specification	Synthesis Result
Latency (μseconds/edge)	0.9	0.53
Throughput (edges/second)	1,111,111	1,886,792
FPGA Area LUTs	274,080	185,663

Table 3.5: **Synthesis Results for HB-C with $\epsilon = 4$**

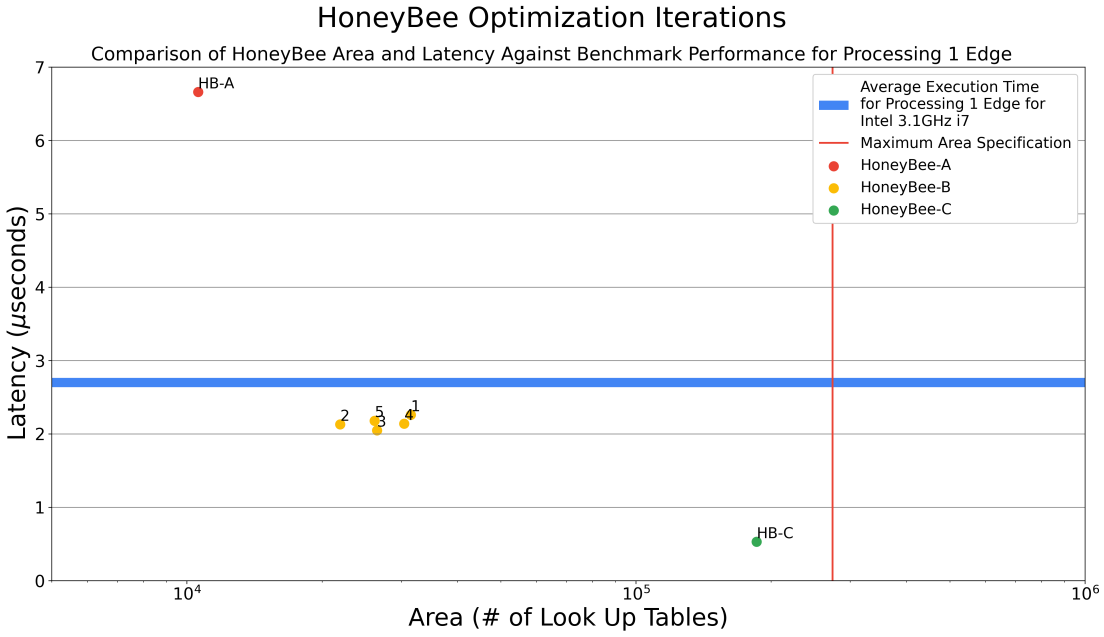


Figure 3.15:

Chapter 4

Motion Planning Architecture

To recap, the bottleneck function of a common motion planning algorithm, RRT, was identified as edge collision detection. The functional hardware unit, HoneyBee, successfully accelerated this function by almost 5 times. The last two objectives of this thesis were:

- To define a RISC-V Extension Instruction Set for the purposes of accelerating motion planning.
- To verify the RISC-V Extension and the functional hardware unit in a complete RISC-V Processor.

4.1 Computer Architecture Background

Computer Architecture encompasses the design of the Instruction Set Architecture and Microarchitecture of a computer. The Instruction Set defines an abstract model of a computer, i.e. how it behaves. Microarchitecture is the implementation of this abstract model, i.e. desiging a CPU to execute the behavious specified in the instruction set.

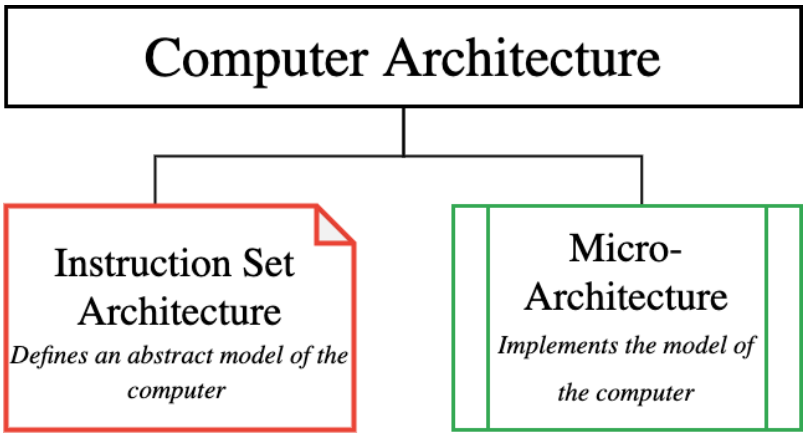


Figure 4.1: **Overview of the Field of Computer Architecture**

4.1.1 Instruction Set Architecture

An Instruction Set Architecture (ISA) is 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.
In more human terms, it can 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.

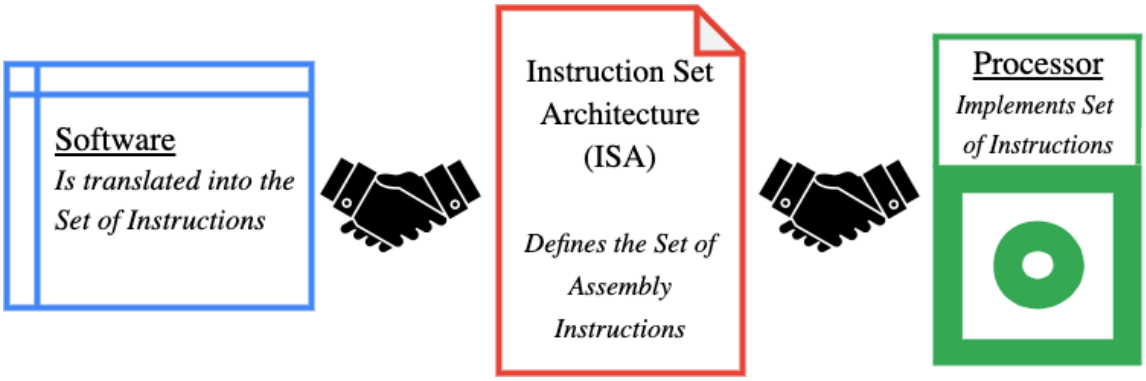


Figure 4.2: **ISA as a Contract Between Software and Hardware Developers**

4.1.2 Microarchitecture

Intstructions and Registers

4.1.3 Reduced Instruction Set Computer

There are two broad classifications of ISAs: Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). Table **??** outlines the key differences between the two.

CISC	**RISC**
Emphasis on Hardware Implementation	Emphasis on Software
Multi-cycle, complex instructions. Different Instructions take different amounts of time to execute.	Single-cycle, simple instructions. All base instructions take the same amount of time to execute.
Operations can be performed directly on values stored in memory.	Memory must be loaded into registers, operated on, and then stored back into memory.
Higher number of cycles per second	Lower number of cycles per second
Smaller Assembly code sizes	Larger code sizes

Table 4.1: **Comparison of CISC and RISC ISAs.**The operating philosophy of the two can really be broken down as follows: CISC has more complex instructions, higher cycles per second, and more cycles per instruction. RISC has fewer, more simple instructions, fewer cycles per second, and generally only one execution cycle per instruction.

Figure **??** shows the most simple layout of a 5-stage RISC Datapath. In the **Instruction Fetch** stage, the processor gets the next instruction from memory for it to be decoded in the **Instruction Decode** stage. Here, the instruction is split into its constituent parts and has certain minor operations performed that are neccesary for the next stage. The **Execution** stage is where most computation occurs. This is where the Arithmetic Logic Unit (ALU) resides, and the result of this computation goes to the **Memory** stage. This is where values are stored into or loaded from the processor's memory, these values, or the values from the Execution stage, are saved to one of the processor's registers in the **Writeback** stage.

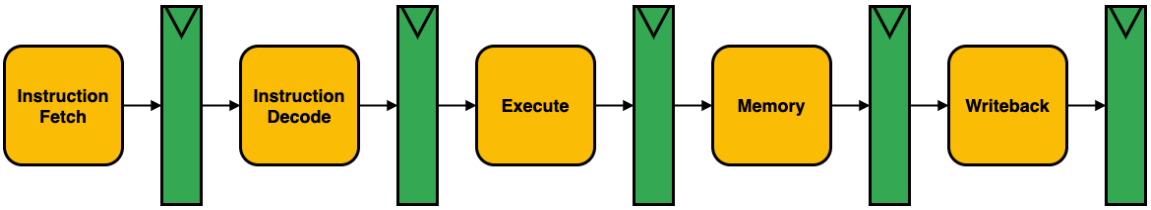


Figure 4.3: **5-Stage RISC Datapath**. Combinational logic stages are shown in yellow, and registers between stages are shown in green.

4.2 RISC-V Instruction Set

4.2.1 RISC-V

(Prounounced "risk-five") is an open-source and extendible ISA developed by the University of California, Berkeley. It is established on the principles of a RISC, a class of instruction sets that allow a processor to have fewer CPI than a CISC (pronounced "risk-five") was developed at the University of California, Berkeley. It is established on the principles of RISC as an open-source and extendable ISA for research and education. It was designed with application specific processors in mind, as they developed a highly flexible and extendable base ISA around which research and acceleration efforts could be based.

The motiviation behind designing RISC-V was largely due to the disadvantages of commercially popular ISAs.[**?**] (Following list adapted from RISC-V ISA Manual).

- **Commercial ISAs are proprietary.** Owners of commercial ISAs carefully guard their intellectual property and will not share implementations.
- **Commercial ISAs come and go.** Many once-popular commercial ISAs have since fallen out of fashion or are not even in production any more. Lingering intellectual property issues interefere with the ability of third-parties to continue supporting the ISA. While an open source ISA may also lose popularity, interested parties can continue to use and support the ISA without interference.
- **Popular commercial ISAs were not designed for extendibility.** There exist almost no ISAs that support extendibility for general purpose computing systems, allowing for no application specific optimizations at the instruction set level.

The overall design of RISC-V can be broken down into 3 characteristics that address the aforementioned limitations of commercial ISAs: Open-source, extendibility, and modularity.

Open-Source

Open-source refers to software of which the owner has granted permission for anybody to study, alter, and distribute the software for any purpose. Often, this means projects are developed in a collaborative public manner. What this means for and ISA is that the RISC-V implementations are often publically available and improved on by developers for their own purposes. Building a high-performance processor from scratch is an arduous, expensive project. Open-source implementations allow for developers to build off existing implementations without all the legwork of implementing a processor from scratch.

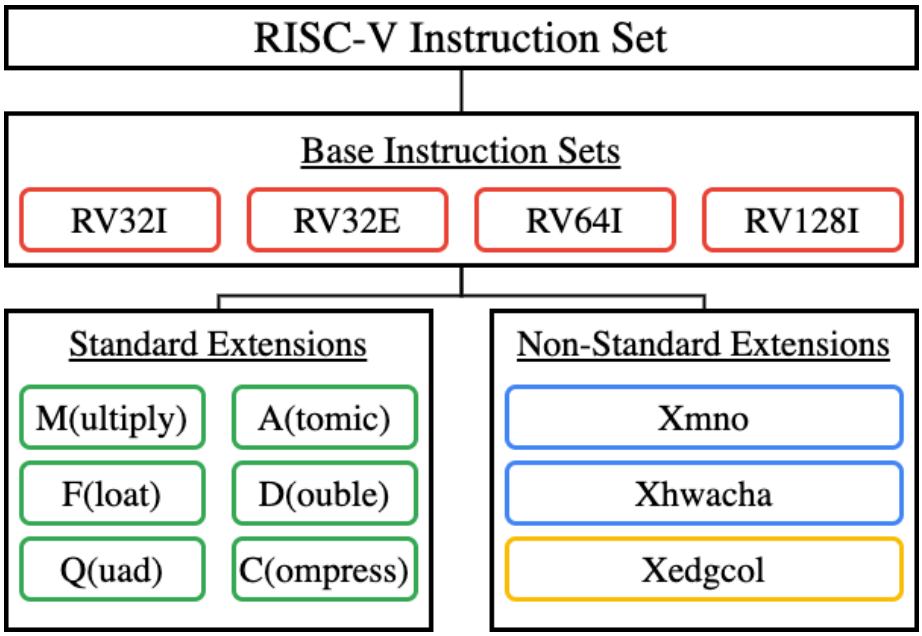


Figure 4.4: **RISC-V ISA Modularity**. It is clear that the RISC-V ISA is in fact a family of ISAs. Each is based on one of the "base" ISAs, which are each the smallest possible ISA to run just about any program. For more instructions and thus, better performance, any number of Standard or Non-Standard Extensions can be implemented on top of a base ISA.

Extendability

RISC-V is designed to be extendable. This means that developers can add their own instructions to the Instruction Set and implement those new instructions in a processor. This processor should still be able to run all other RISC-V compiled programs, along with a set of programs for which it is specially optimized. This level of backwards compatibility means that an extended ISA is not just a new ISA.

Modularity

Finally, RISC-V was designed to be relatively easy to implement. It is broken down into small "base" ISAs which can then have any number of standard and custom extensions added to them. These base ISAs are not designed in such a way to overarchitect for a certain type of microarchitecure. This modularity and flexibility is part of what makes it such an attractive proposition for specialized computer architecture. Figure **??** demonstrates this modularity.

4.2.2 RV32I

The RV32I (Short for RISC-V 32-Bit Integer) base ISA is a small ISA of only 40 unique instructions, but sufficient to support modern operating systems. It has 32 registers, `x0-x32`, each 32-bits wide. `x0` is a hard-wired 0, and there is also another register dedicated for the program count.

Move some of this to appendix. This bit needs work

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

> 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 **??** 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 4.2: Register State for RV32I Base Instruction Set

Instruction Formats

Table **??** 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 4.3: RV32I Base Instruction Formats

4.2.3 Defining a RISC-V Custom Extension

Extension Specifications

Talk about determining honeybee output

Table **??** specifies that the edge collision unit takes an edge e with a maximum length of ϵ. When implementing HoneyBee, it was necessary to determine a fixed value for ϵ.

Defining Extension

4.3 PhilosophyV

Philosophy 4, written in 1903 by Mr. Owen Wister of the Class of 1882 (founder of the Western literary genre), recounts the antics of two Harvard students and their last minute attempts to study (or avoid studying) for a Philosophy exam for which they are hopelessly unprepared. Similarly, this section details the process of building a RISC-V processor, by far the most intricate engineering challenge of this Thesis, and a task for which I was unsure of my preparedness. As such, this processor was named **Philosophy V**; both in reference to the RISC-V ISA for which it was 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 **??** provides a schematic of the PhilosophyV processor.