If Going to Crash: Don't. RISC-v Architecture for Motion Planning Algorithms in Autonomous UAVs

IF GOINGTOCRASH: DON'T

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

April 10th, 2020
Version: 5.0

Abstract

This thesis presents an extension for the RISC-V Instruction Set Architecture (ISA) for the purpose of faster motion planning in autonomous Unmanned Aerial Vehicles (UAVs). Fully autonomous UAVs have the potential to change the world in which we live, but they are currently unable to pilot themselves in high-complexity, obstacle-dense environments; The processors that they employ cannot execute motion planning software quickly enough. RISC-V is a relatively new ISA that is founded on the principles of open-source and extendibility, making it an excellent ecosystem for designing application specific processors. However, as of April 2020, no attempts had been made to develop motion planning architecture within the RISC-V ecosystem.

This thesis serves as a proof-of-concept for accelerating motion planning with RISC-V architecture. It presents the implementation of "HoneyBee", a microarchitectural unit that can compute collision detection over 5 times faster than a general-purpose Intel CPU. More significantly, it defines a motion planning extension for RISC-V that simplifies the number of instructions required to detect an edge collision from hundreds of thousands to only one. These promising results demonstrate the viability of the approach, and should encourage developers to embrace RISC-V in the development of motion planning processors for autonomous drones.

Dedication

To my parents, who have always encouraged my academic pursuits.

Acknowledgments

To my faculty advisor, Prof. Vijay Janapa Reddi, thank you for your guidance and for inspiring my passion for computer architecture in my Junior year.

To Jon Cruz, I couldn't have done this without you. Thank you for introducing me to the tools I needed, for your invaluable advice, and for always being available.

To Patrick Ulrich and the members of my ES100 section, I appreciated our weekly meetings and the fresh perspectives you offered.

To Isabella Rhyu, thank you for your help with the more mathematically rigourous parts of this thesis, though I know you found them easy.

To Mr. Trombetta, Jeffrey and Balmore. I thank you with heart and soul for providing a home to me for the past three weeks. I regret that our time together was cut short, but I look forward to coming up again soon - I'm sure it will be the same forever.

Contents

List of Acronyms

2D 2-Dimensional

3D 3-Dimensional

ALU Arithmetic Logic Unit

API Application Programming Interface

ARM Advanced RISC Machine

ASP Application Specific Processor

CISC Complex Instruction Set Computer

CPU Central Processing Unit

CSV Comma Seperated File

DOF Degree-of-Freedom

FPGA Field Programmable Gate Array

GPU Graphics Processing Unit

HB-A HoneyBee-A

HB-B HoneyBee-B

HB-C HoneyBee-C

HDL Hardware Description Language

HLS High Level Synthesis

IEEE754 IEEE Standard for Floating-Point Arithmetic

ISA Instruction Set Architecture

LSB Least Significant Bit

LUT Look-Up Table

MSB Most Significant Bit

OGM Occupancy Grid Map

PRM Probabalistic Road Map

RISC Reduced Instruction Set Computer

RRT Rapidly-exploring Random Tree

RRT* Rapidly-exploring Random Tree Star

RTOS Real-Time Operating Systems

RV32I RISC-V 32-Bit Integer

SoC System on Chip

UAV Unmanned Aerial Vehicle

List of Algorithms

List of Figures

List of Tables

Glossary of Terms

A priori: relating to or denoting reasoning or knowledge which proceeds from theoretical deduction rather than from observation or experience.

Application Specific Processor (ASP): A computer processor that has been optimized for a specific function or set of functions that support a given application.

Arithmetic Logic Unit (ALU): A combinational circuit of a computer processor that performs basic mathematical calculations on a pair of inputs.

Automata: moving mechanical devices made in imitation of human beings.

Axis-oriented plane: Planes that run parallel to one of the xy plane, the xz plane, or the yz plane.

Bit: A binary digit, 1 or 0, and the most basic unit of information in computing. Bits are combined to represent complex information. Every single thing your computer does is, eventually, represented and implemented in the movement of bits.

Bit-width: The number of binary digits necessary to represent a data type. For instance, a boolean takes 1 bit (1 or 0), whereas an integer may be represented by any number of bits (most commonly 32).

Complex Instruction Set Computer (CISC): A computer in which single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) or are capable of multi-step operations or addressing modes within single instructions.

Configuration: A specification of a robot's location, position, and setting in a space. For example, when a robot is represented by a single point in 3D space, its configuration is merely x, y, and z coordinates. But if a robot is represented as a 3D humanoid with a head, body, arms and legs, then its configuration would be its position in 3D space, its orientation in 3D, and the position of all its joints and limbs such that the space being taken up by the robot can be exactly determined. Consequently, as the physical complexity of a robot increases, so too does the complexity of representing it algorithmically.

Degree-of-Freedom (DOF): Refers to the number of independent factors that describe the configurations in which a robot can exist and motion can occur.

Dijkstra's algorithm: An algorithm for finding the shortest path between two nodes in a graph.

Field Programmable Gate Array (FPGA): An integrated circuit designed to be configured by a designer after manufacturing – hence the term "field-programmable". Its behaviour is specified in software, using a Hardware Description Language (HDL).

Hardware acceleration: The process of speeding up the execution of a function by implementing part or whole of that function specifically in hardware.

Hardware Description Language (HDL): A computer language used for designing computer hardware. It is used to define the behaviour of modules, simulate their performance, and synthesize them on a Field Programmable Gate Array (FPGA).

High Level Synthesis (HLS): An automated hardware design process that takes software written in high-level languages (often C, C++) that algorithmically defines a function, and converts that into an HDL that implements that function.

IEEE Standard for Floating-Point Arithmetic (IEEE754): The technical standard for how floating-points are represented and processed in binary, established by the Institute of Electrical and Electronics Engineers.

Instruction Set Architecture (ISA): An abstract model of a computer, that defines the instructions, registers, memory, behaviour, and other attributes of a computer architecture. It can be thought of as the contract between software and hardware developers, as the ISA lists the instructions that software may be implemented in, and the instructions that a processor must support.

Look-Up Table (LUT): 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.

Mathematically complete: An algorithm is mathematically complete if it will always find all solutions.

Occupancy Grid Map (OGM): A method of representing a 2-Dimensional (2D) or 3-Dimensional (3D) space by dividing it into discrete "grids" and marking each whole grid as "occupied", even if only part of it is.

Pragma: From the word "pragmatic". A programming directive that specifies how the relevant code should be processed. In the context of High Level Synthesis (HLS), they are the directives that can be used to optimize how C Code is synthesized into HDL.

Probabalistic Road Map (PRM): A motion planning algorithm that randomly samples free space, and then connects sampled configurations with nearby configurations to build a map.

Probabilistically complete: Describes an algorithm with a likelihood of finding a solution that approaches one as its runtime approaches infinity.

Rapidly-exploring Random Tree (RRT): 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.

Reduced Instruction Set Computer (RISC): A computer architecture based on a small number of instructions executed in a small number of cycles.

RISC-V: (Prounounced "risk-five") is an open-source and extendible Instruction Set Architecture (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 Cycles Per Instruction (CPI) than a Complex Instruction Set Computer (CISC).

RISC-V 32-Bit Integer (RV32I): One of the four base ISAs within RISC-V. While it implements integer values only in 32-bit representations, it contains the minimal number of instructions for a fully working computer processor.

Time complexity: Refers to the amount of time taken by an algorithm to run as a function of the length of its input.

Unmanned Aerial Vehicle (UAV): An aircraft without a human pilot on board. It may be piloted remotely completely by a human pilot, autonomously pilot itself, or a mixture of the two.

Workspace: The space which a robot and obstacles occupy in motion planning problems.

Chapter 1

Introduction

If only it were that simple... The above image is an exchange posted on the subreddit "Programmer Humour", satirizing the gross oversimplification of the deep complexity of fully autonomous drone flight. This thesis studies the "coding and algorithms" of the topic extensively, from the micro-implementation of specialized motion planning processors to the high-level software that could, theoretically, allow a programmer to write:

```
if goingToCrash: dont();
```

1.1 Problem Summary

1.1.1 Background & Motivation

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* [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* (prior to execution).

In recent years a new generation of fully autonomous robots has been developed for a wide range of complex applications. A specific case is the autonomous Unmanned Aerial Vehicle (UAV). The UAV has been utilised in military applications extensively throughout the late 20th and early 21st century. Only recently has their potential in industry begun to be realised. As technology improves, UAVs are moving from remote piloting to being able to pilot themselves without human control. These autonomous UAVs are required to pilot themselves through complex, ever-changing environments. This means executing motion planning software.

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 2 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 execute too slowly on regular Central Processing Units (CPUs) for autonomous UAVs to operate in rapidly changing, 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).

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), commerical Instruction Set Architectures (ISAs) are not designed for extendability, meaning that even a highly specialized processor is limited by general purpose instructions.

RISC-V

RISC-V (pronounced "risk-five") is an ISA developed by the University of California, Berkeley. It is established on the principles of modularity, extendibility, and open-source contribution. RISC-V was started with the philosophy of creating a practical ISA that was usable in any hardware or software without royalites. The first report describing the RISC-V Instruction Set was published in 2011[2].

Due to its flexibility, it has excellent potential in the space of application specific processors. However, as of April 2020, no attempts have been made to develop a motion planning processor using the RISC-V ISA. As such, this thesis serves as a proof-of-concept for this neglected opportunity.

1.1.2 Problem Definition

Problem Statement

Motion planning software running on general purpose CPUs cannot execute quickly enough for fully autonomous UAVs to operate in high-complexity environments. The common strategy of using power-hungry GPUs to accelerate the execution of this software requires too much power to be feasible for UAVs to sustain flight for useful periods of time. While there are significant barriers to designing application specific processors, RISC-V presents an excellent, but neglected, ISA with which specialized motion planning processors can be implemented.

Intended Audience

This thesis has been written such that it can be understood by those with no background in computer architecture. However, the project was conducted with computer architects in mind, with the aspiration of proving the merits of the RISC-V ISA in designing motion planning specific processors.

1.2 Project Overview

1.2.1 Project Goals

This is not the first project to attempt to accelerate motion planning algorithms using hardware; motion planning optimization has been studied extensively in hardware and in software. But the problem persists that, due to the nature of conventional ISAs, there are significant barriers to implementing motion planning specific processors.

RISC-V was released 9 years ago. While it is starting to gain some traction in the commercial space, RISC-V has not yet been used to develop a motion planning specific processor. As such, the overall mission of this thesis is as follows:

Mission
To present a proof-of-concept for extending and implementing RISC-V to develop motion planning specific processors.

To achieve this, the following 4 objectives must be accomplished:

Project Objectives

1. Determine the computational bottleneck of a commonly used motion planning algorithm.
2. Implement a functional hardware module to replace the bottleneck function.
3. Define a motion planning extension to the RISC-V ISA
4. Build a fully functional motion-planning processor that implements the extended RISC-V ISA

Computer Implementation Hierarchy

To briefly frame the space in which this thesis operates, consider the hierarchy of computer implementation, demonstrated in Figure 1.1. **User level applications**, such as Microsoft Word, Google Chrome, and Apple's iTunes, sit at the top of the hierarchy. These applications are implemented in **High/Mid Level Languages**, such as C/, C++, Python, Java, etc. This software is eventually executed on a **Processor**, otherwise known as a CPU. A processor does not understand Python, or any other programming language for that matter. The only thing it can understand is binary values (a one or a zero). Every computer program gets translated into ones and zeros (called machine code) for execution on a processor. But how are languages like C translated into machine code? The layer between programming languages and the processor is the **Instruction Set Architecture (ISA)**. Chapter 4 goes into detail about ISAs and how they work. For now, if unfamiliar with the topic, it is sufficient to think of an ISA as a translator between programming languages and machine code. This thesis operates across the lower three levels of this hierarchy.

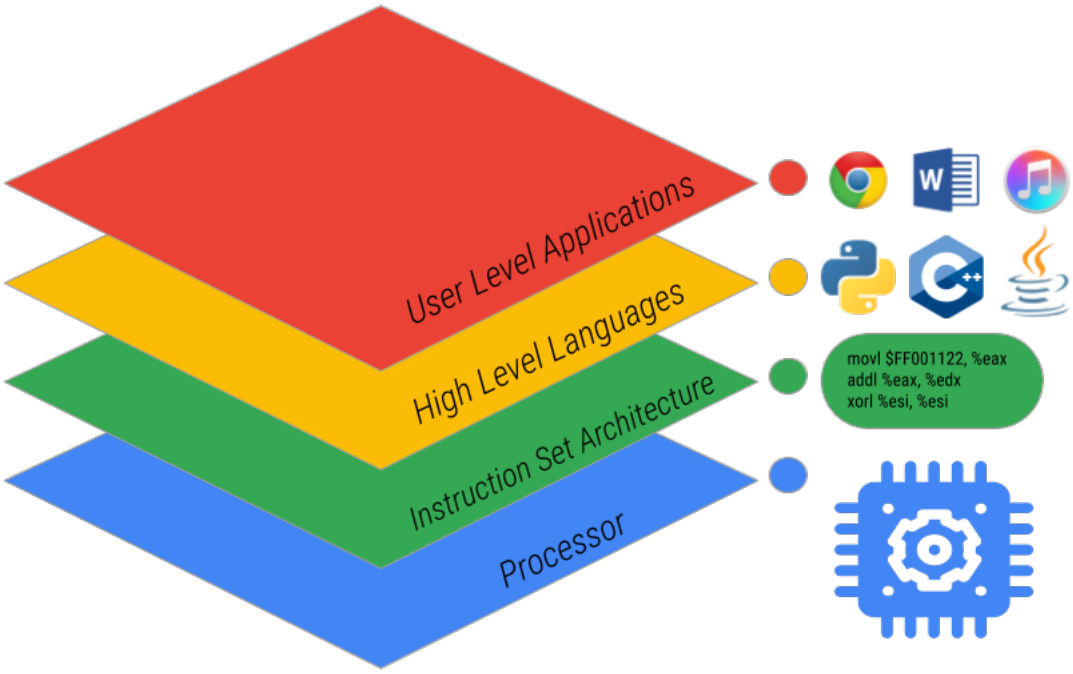


Figure 1.1: **Simple Visualization of Computer Implementation Hierarchy**

1.2.2 Project Structure

Chapter 2 details how Rapidly-exploring Random Tree (RRT), a commonly used motion planning algorithm, is implemented and analysed. From this analysis, it was determined that the computational bottleneck is edge collision detection. Chapter 3 outlines a process for implementing this bottleneck function in a hardware module, achieving a 5× speedup. Chapter 4 defines the RISC-V motion planning extension and describes the build of a RISC-V processor that implements this extension and the Honeybee unit.

System Overview

Figure 1.2 shows a high level overview of the system this thesis proposes.

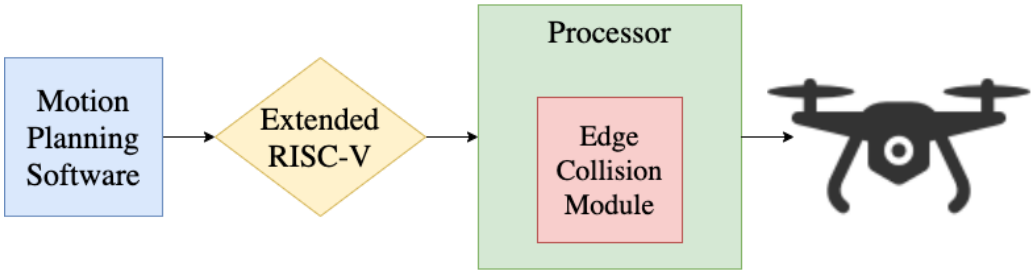


Figure 1.2: **System Diagram of Overall Project**. Motion planning is implemented in software, compiled into machine code through the extended RISC-V ISA. This is then executed on the drone's processor that includes the motion planning accelerator.

Table 1.1 on Page 8 outlines the components of this system and their descriptions.

Measure of Success

This thesis will be considered a success if it can present a compelling argument for the merits of designing motion planning specific processors **with the RISC-V ISA**. This will require:

- Demonstrating a clear performance limiting bottleneck function of motion planning algorithms.
- Achieving significant performance improvements by implementing the identified bottleneck function in hardware.
- Proving that a custom RISC-V extension can be easily defined and clearly reduces the complexity of executing the bottleneck function.

Component	**Source**	**Description**
C-Implementation of RRT		
Rapidly-exploring Random Tree	*A.J.W. Kenny*	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
RISC-V Instruction Set		
RISC-V 32-Bit Integer (RV32I)	Berkeley	40 Instructions defined such that RV32I is sufficient to support modern operating systems [3].
Motion Planning Extension	*A.J.W. Kenny*	This is the custom extension defined by this thesis targeting motion planning instructions. It is outlined in Chapter 4.
Motion Planning Specific Processor		
PhilosophyV	*A.J.W. Kenny*	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	*A.J.W. Kenny*	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**

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.

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*[4], and by 1997 the *Deep Blue* computer defeated Garry Kasparov, the reigning world champion, in a six game chess match.[5] 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

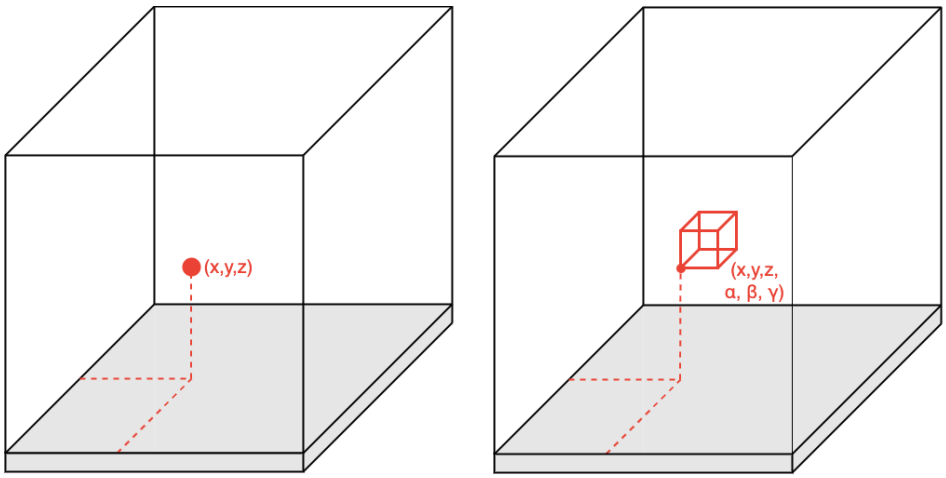
Workspace

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 and orientation of the robot. The complexity of a robot's configuration is 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 2D space and (x, y, z) in 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 OGM will more accurately represent a workspace with a higher resolution, shown in Figure 2.2.

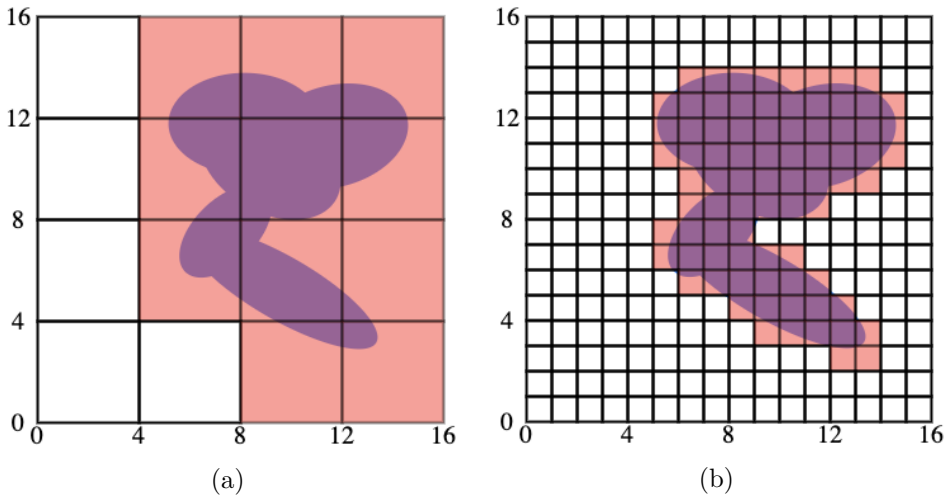


Figure 2.2: **Occupancy Grid Maps for a (16×16) Workspace of Different Resolutions**. Figure 2.2a shows how an OGM with low resolution, while simpler to construct and analyse, will over-represent the obstacle density of a workspace. Figure 2.2b shows how a higher resolution will more accurately reflect 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[6] and J. Kuffner[7]. 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 2.3. **It does not calculate the fastest path from that tree**; that can be accomplished using algorithms such as Dijkstra's algorithm.

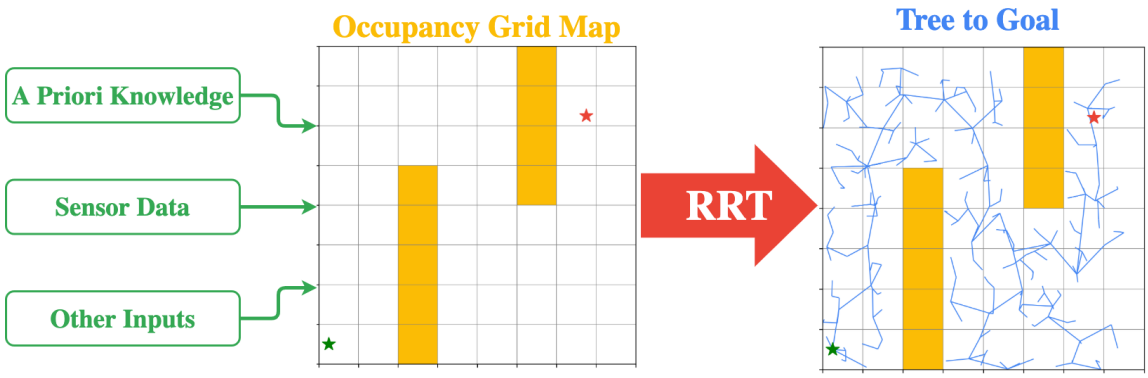


Figure 2.3: **Scope of the RRT Algorithm**: Takes an OGM as input and outputs a tree of collision free paths. The tree is shown in blue on the right.

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 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 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 2.1 can be visually represented in Figure 2.4 for a 2D robot.

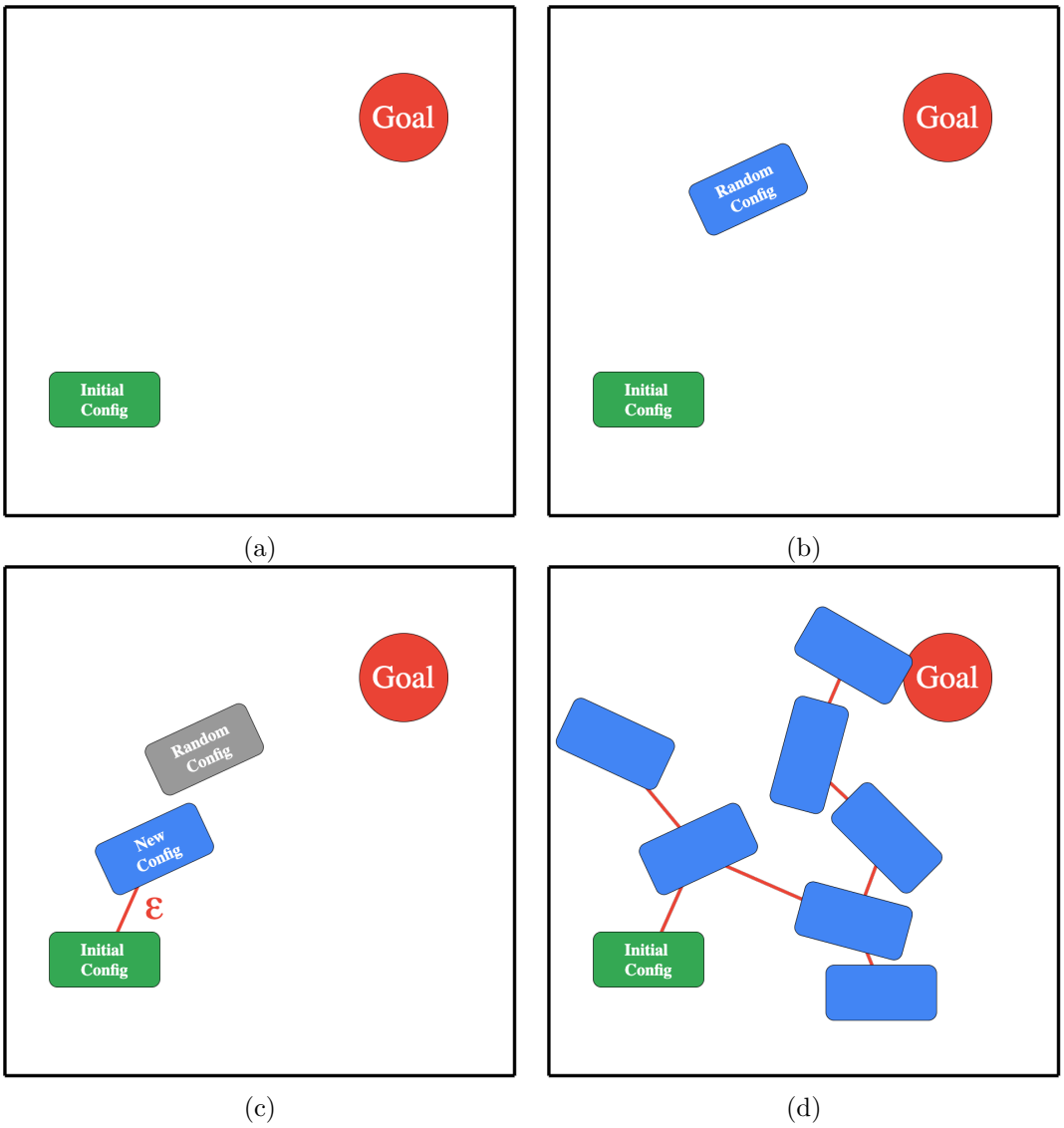


Figure 2.4: **Demonstration of RRT Algorithm for 2D robot in 2D space.** In this example, the graph G begins with an initial configuration and a goal. In (b), the first random configuration is generated. In this case, the nearest configuration is the inital configuration. The random configuration is 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 2.1 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 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's configuration is in the same spot as the obstace, i.e. a position the robot could not physically occupy) 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 2.2. The method of implementing RRT with collision detection to model a drone in 3D space is detailed in Section 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 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 2.1. Appendix B.2 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 of 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 B.1 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 B.3 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 B contains a more detailed account.

Parameterization

Table 2.2 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

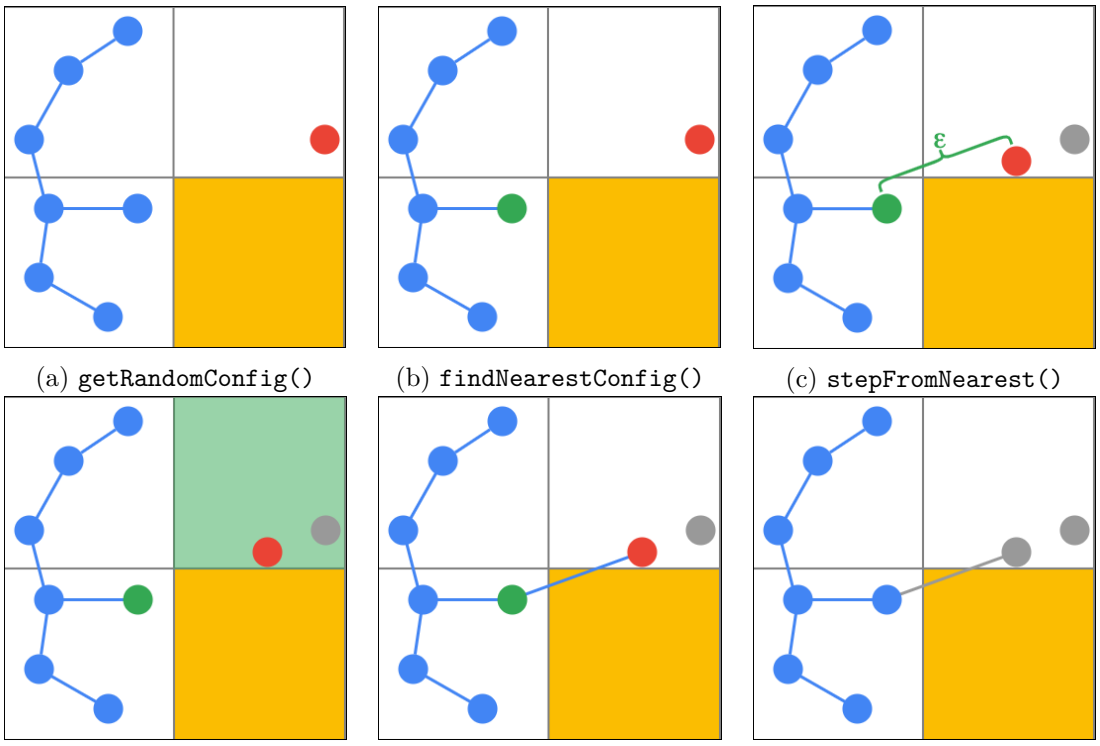
RRT was implemented in both 2D and 3D. Not only did a 2D implementation provide a good development checkpoint, it was also interesting to see the difference in computational load between 2D and 3D, shown in Section 2.3.

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 2.2 shows that there are 5 key functions that constitute RRT. Figure 2.5 demonstrates each of these functions: `getRandomConfig()`, `findNearestConfig()`, `stepFromNearest()`, `configCollisions()`, and `edgeCollisions()`. Appendix B.4 shows in detail how each of these functions was implemented.



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

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

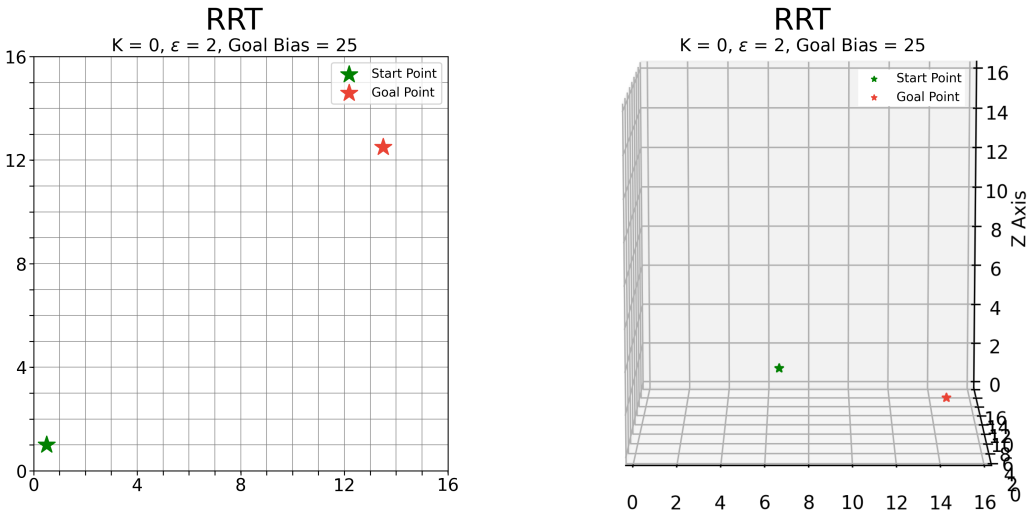
Figure 2.5: **Demonstration of the 5 Key Functions that Constitute RRT**, where configuration is a node in a 2D workspace. A new node is generated in (a) with `getRandomConfig()`, and the closest existing node is found with `findNearestConfig()` in (b). In this case, the new node is further than ϵ from 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 2.6. 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. **Note:** The 2D space and 3D space have different start and end points and different OGMs to better demonstrate how RRT functions.



(a) Workspace in 2D

(b) Workspace in 3D

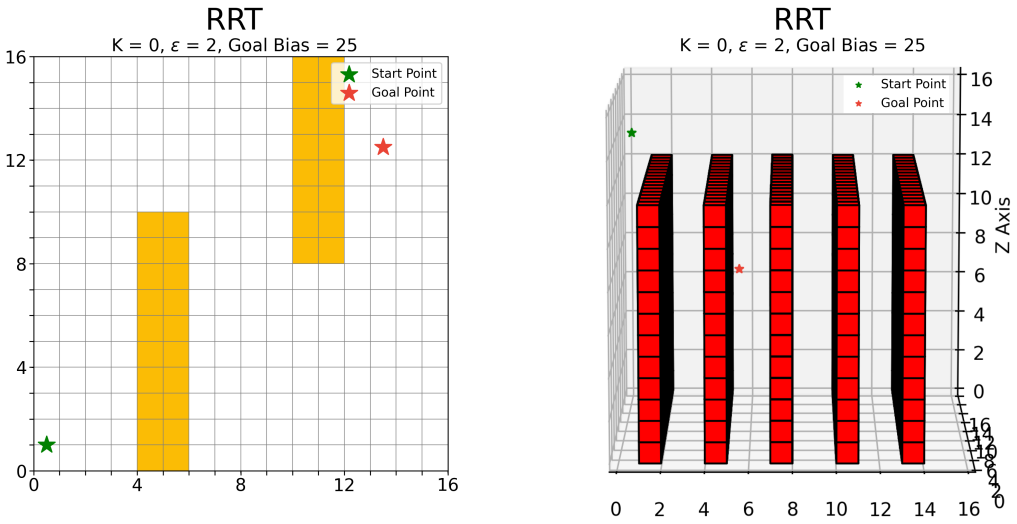
Figure 2.6: **Visualization of Workspace in 2D and 3D**, with configuration represented by only a point, and Start and Goal nodes shown

Plotting Obstacles

Obstacles were plotted in accordance to the input OGM, shown in Figure 2.7

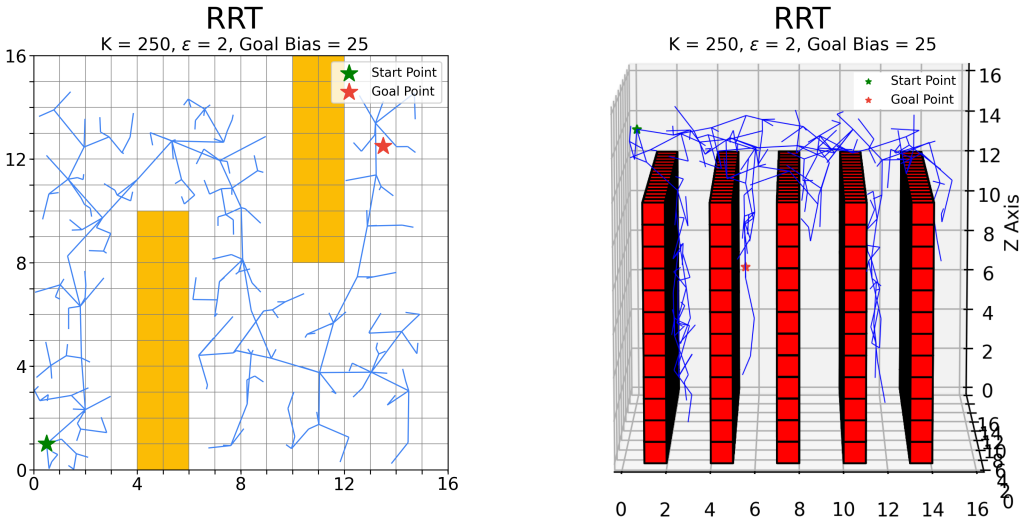
Plotting RRT Graph

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 2.8



(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 2.1, 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 take 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 2.3.1.

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[8]. 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 B.6 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 running RRT 100 times for each set of parameter values. 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 2.3.

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 B.3 on in appendix B.7. This was to be expected. The more significant results are the breakdowns of CPU time for each function, detailed on the following 2 pages.

2D Computational Load Profile

Figure 2.9 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.

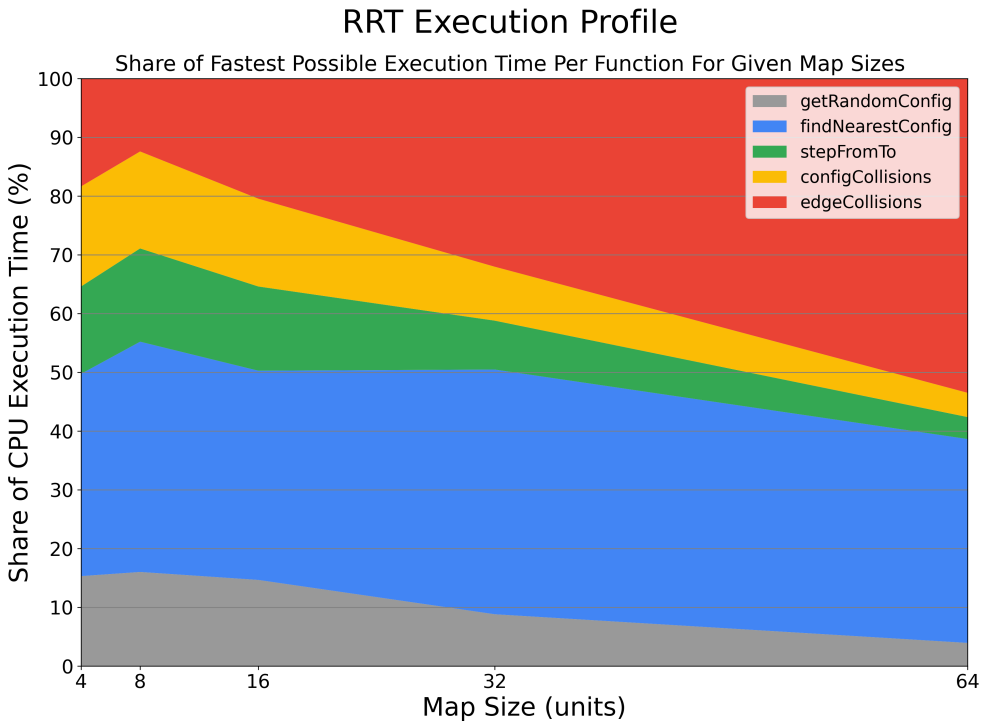


Figure 2.9: **Profile of Computational Load of RRT in 2D**

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 2.10.

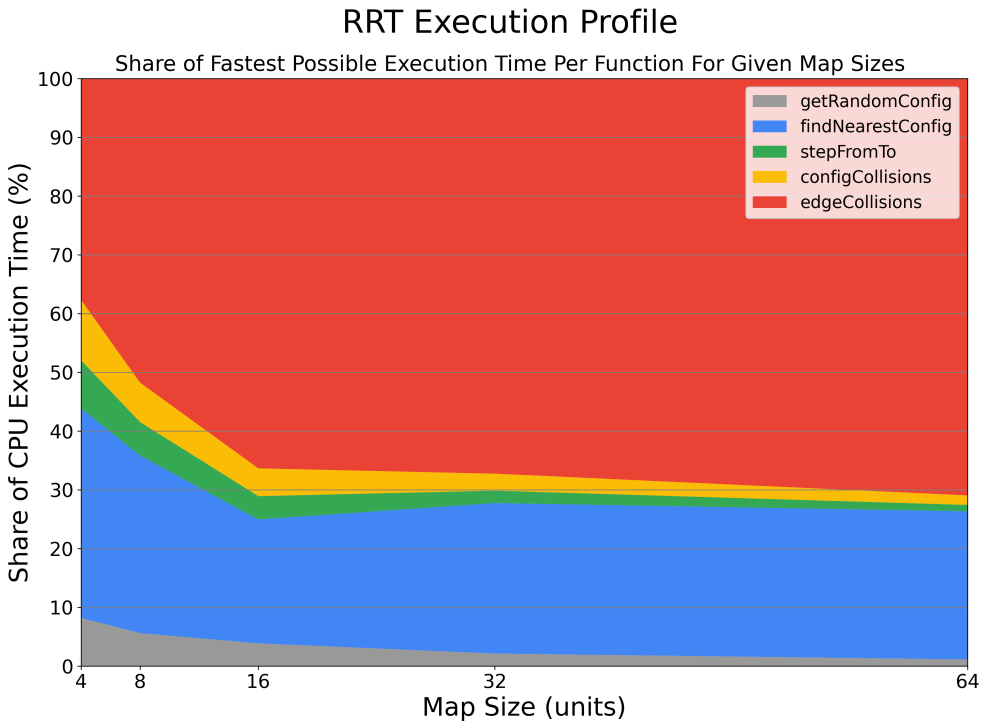


Figure 2.10: **Profile of Computational Load of RRT in 3D**

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 that 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 intersected are "occupied", a collision is returned. This is shown in Figure 3.2 on Page 26.

Calculating intersections between a segment and grids is very computationally intense. This is because it is a fairly involved geometric process. Figure 3.1 on Page 25 shows how grid intersections are detected by computing the point at which the segment intersects certain **axis-oriented planes**.

Time Complexity

With the steps of the edge collision algorithm understood (explained graphically in Figure 3.1, algorithm included in Appendix B.4), 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 $O(3\epsilon)$.

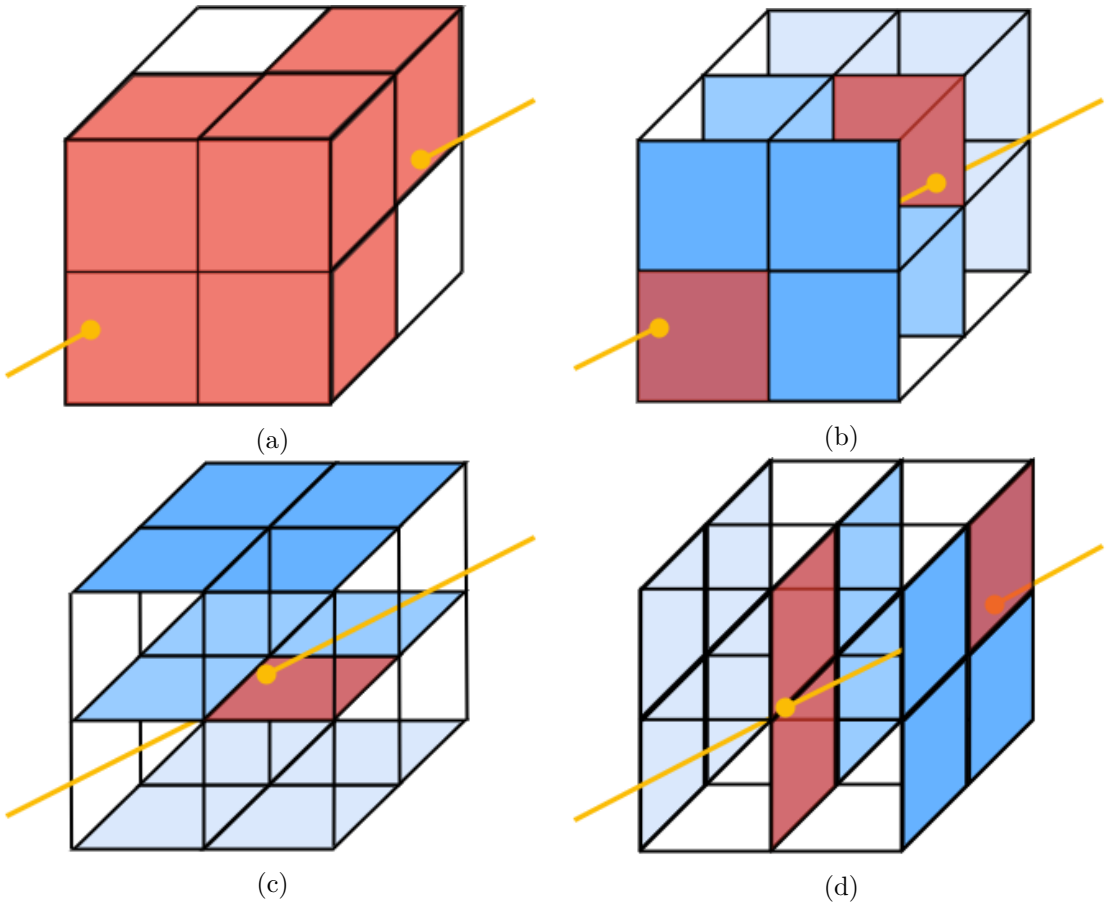


Figure 3.1: **Detecting Grid Intersections by Finding Intersections with Axis Oriented Planes**

Consider an edge e that spans the width of a $2 \times 2 \times 2$ grid map, as shown in Figure 3.1 (do not consider if the grids are occupied, this is just to determine which of them the edge intersects). Just by eyeballing Figure 3.1a it seems odd that so many of the grids have been intersected (denoted by red shading) by the yellow edge. The algorithm executes by checking one set of axis-oriented planes at a time. Figure 3.1a 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 (3.1c). In Figure 3.1d, 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.

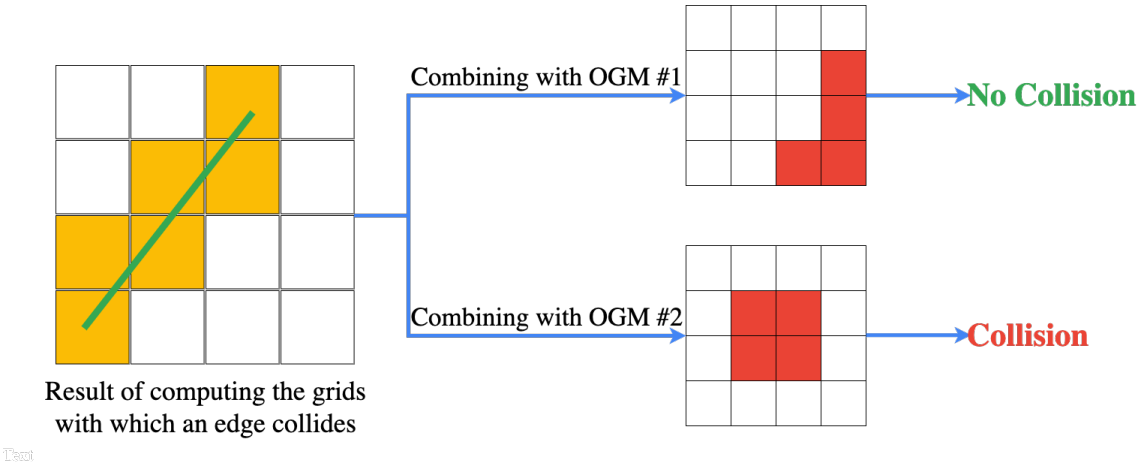


Figure 3.2: **Edge Collision Computation Process**. Most of the computational load is found in determining the grids with which an edge intersects (as seen on the left of the figure). Once these grids have been found, it is very simple (and fast) to lookup if these grids are occupied in the OGM

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 that the edge collision function run fast enough 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 3.1 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 3.2, 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 the intersections found in a $\epsilon \times \epsilon \times \epsilon$ gridspace can be represented as an ϵ^3 sequence of binary values. Table 3.2 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[9] 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 3.4. 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 3.1.1.

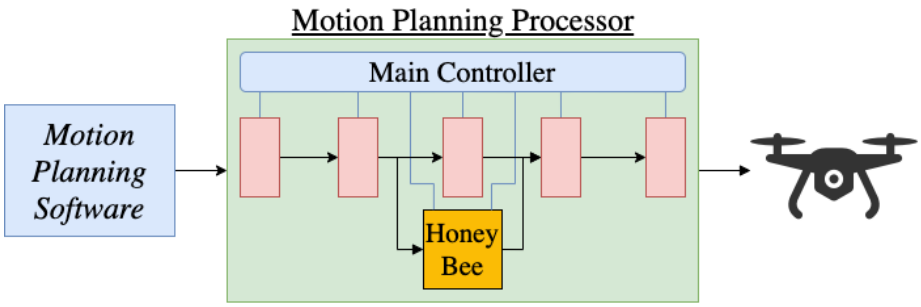


Figure 3.4: **HoneyBee in a Motion Planning Processor**. Shown is an abstraction of how HoneyBee would become an extension of the normal processor datapath.

3.2.1 HoneyBee Interface Design

The interface for the HoneyBee functional unit, following on from the interface specifications outlined in Section 3.1.2 can be simply represented by Figure 3.5.

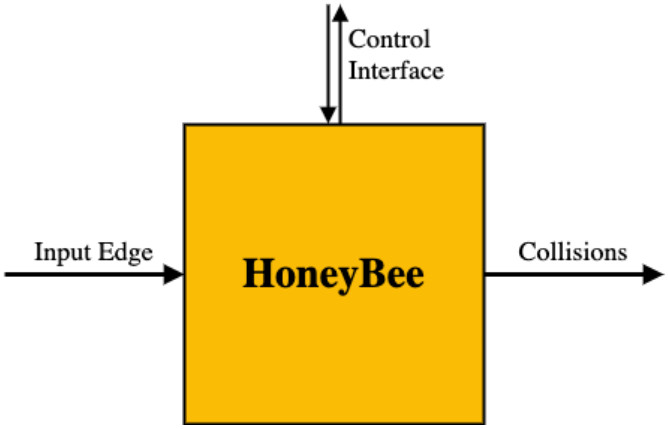


Figure 3.5: **General Overview of HoneyBee Interface**. The functional unit takes an edge e, defined by two points p_1 and p_2, as an input, and outputs a series of collisions. These collisions describe which grids an edge intersects. Its control interface allows for communication with a processor's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

However, when designing hardware (the method of doing so is described in section 3.2.2), how these inputs and outputs are implemented must be considered at the bit level. Figure 3.6 shows all input, output, and control ports, and their bit-widths.

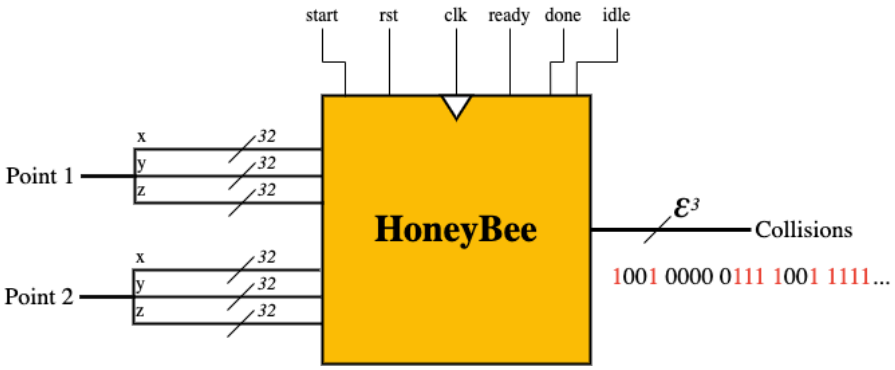


Figure 3.6: **Port Diagram of HoneyBee Interface**. The edge input is represented by 6 32-bit floats, following the IEEE 754 Single Precision 32-bit protocol, with each float representing one of its coordinate points. The output sequence of collisions is an e^3-bit sequence, with each bit in the sequence 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

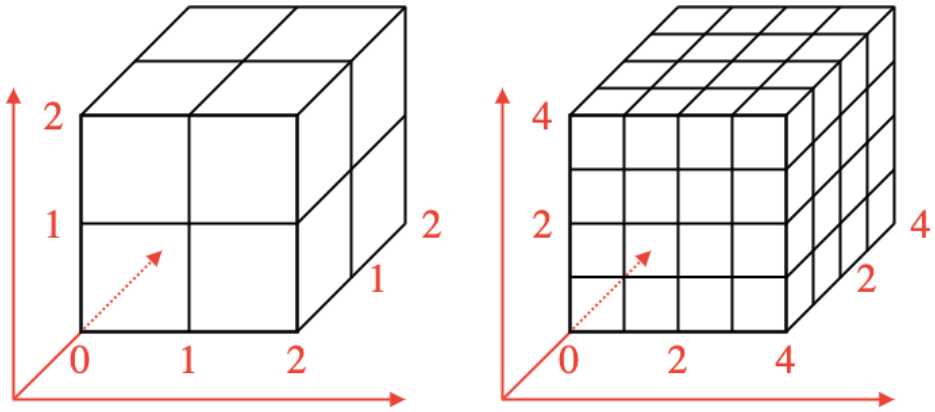
The inputs to HoneyBee collectively describe a single edge. This is done with 6 32-bit floating point numbers. How floating points (non-integer numbers) are represented in binary is defined by the IEEE Standard for Floating-Point Arithmetic (IEEE754). How this is actually represented is not neccesary to understand, but is explained in Appendix C.3. The important point is that the input edge is determined by 6 32-bit coordinate points.

Output

HoneyBee outputs a sequence of "collision-bits," with each bit in the sequence representing any collisions between the input edge with its corresponding grid. How this sequence of bits is mapped to a 3D grid-map is explained in Appendix C.4. 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 possibly intersect.

Since the number of grids being checked is parameterized, so must the number of collision-bits. This is demonstrated in Figure 3.7.

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



(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**

different output bit-widths. When the time comes to add HoneyBee to a processor, it is 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 C.5. 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.[10] 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.[11]

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).

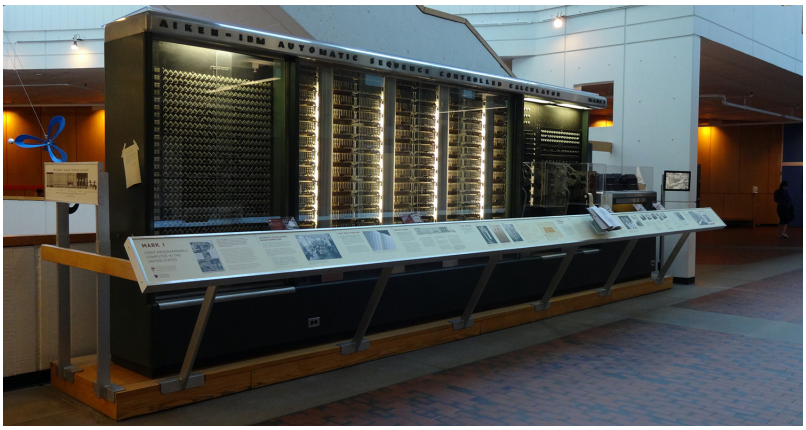


Figure 3.8: **The Harvard Mark I Computer**, photographed in the Harvard Science Center in April 2014 (Source: Harvard University)

HDLs are a family of computer programming languages that are used to specify the function 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 3.9 demonstrates this process.

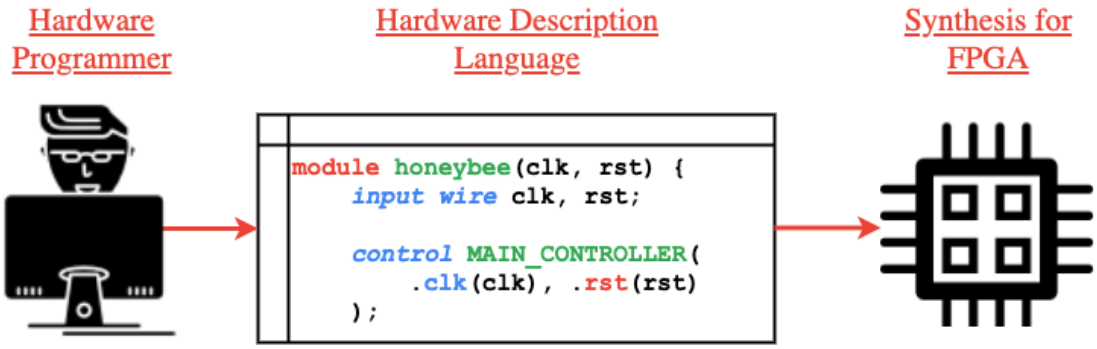


Figure 3.9: **The Hardware Development Process**, Defining Hardware Units in Hardware Description Languages for FPGAs.

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

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.

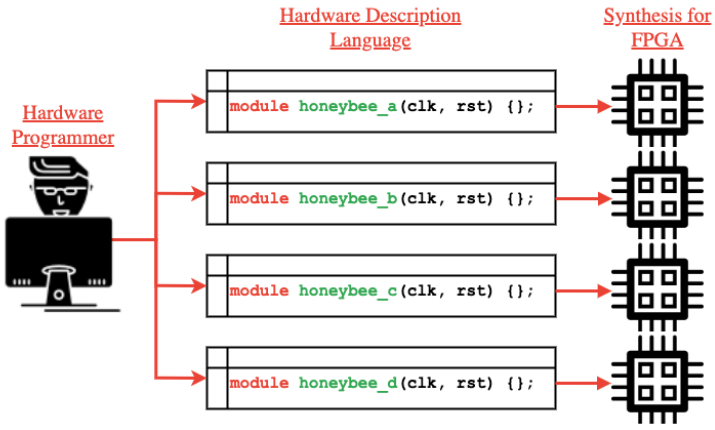
The most important benefit of using HLS, however, is the ability to use "pragmas." These are directives given to the HLS tool that tell it what optimizations to use when translating C code into an HDL. This allows the same funtionality to be synthesized in many different ways, optimizing the synthesis for speed, area, memory, etc. As such, this hardware development process allows developers to experiment quickly with different ways to implement the same functionality. This is demonstrated graphically by Figure 3.10 on Page 34.

HoneyBee-A Synthesis

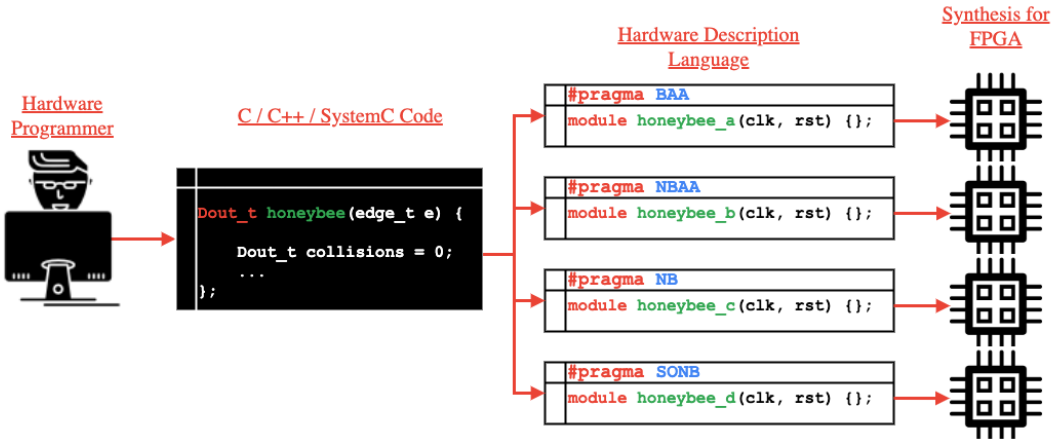
With the functionality of HoneyBee implemented in C, the first optimization iteration (designated HoneyBee-A (HB-A)) was synthesized. HB-A had no pragmas, and was merely a basic hardware implementation of the defined functionality. HB-A (and all subsequent iterations) synthesized correctly and satisfied the interface specifications (See Appendix C.6 for technical interface report of synthesis). Table 3.3 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 comes 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 3.10a shows how, without using HLS, the hardware developer must write multiple different implementation of the same functionality to acheive different performance. Figure 3.10b 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 3.11.

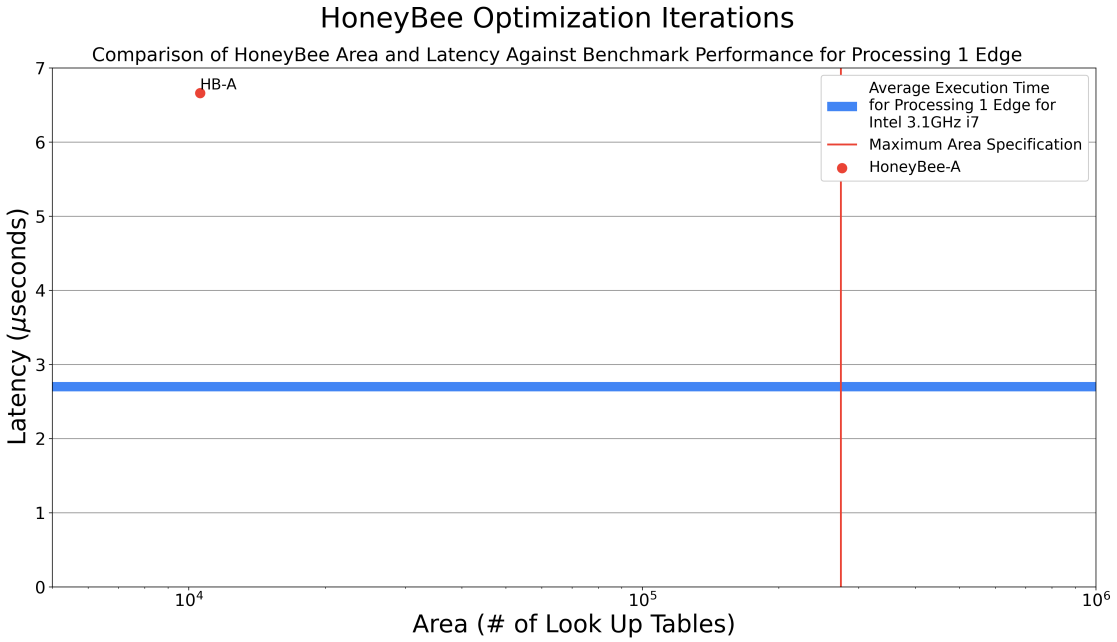
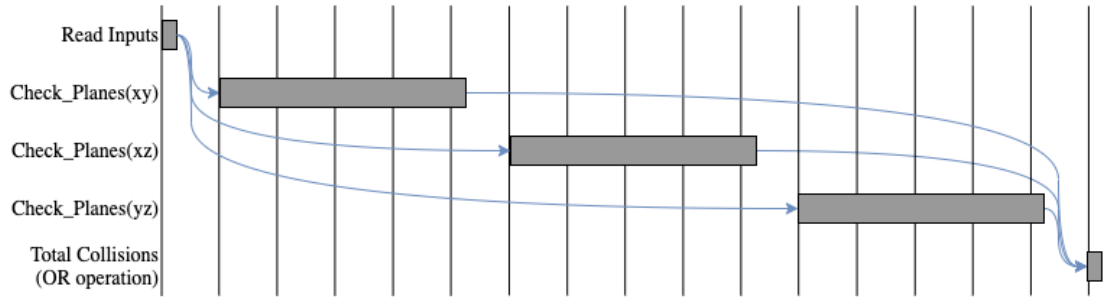


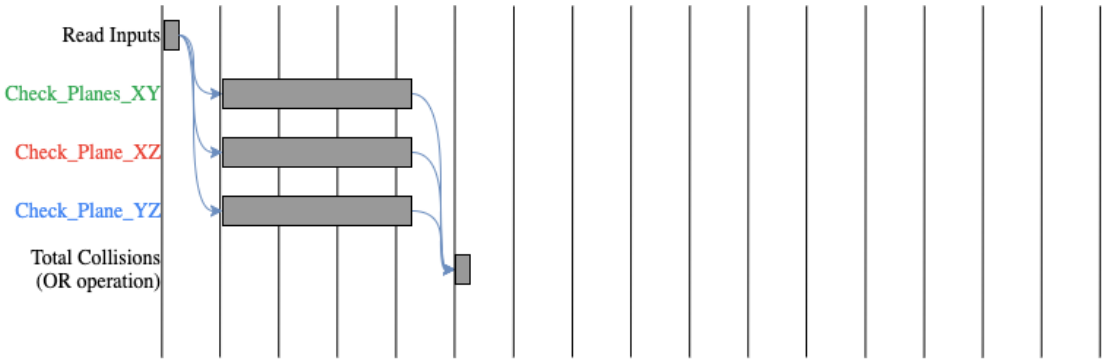
Figure 3.11: **HB-A Performance Against Benchmark CPU**. Standard deviation of CPU average performance is shown by width of the line for easier interpretation.

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 3.12 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 3.12a), 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 3.12b), there are three separate 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 3.12 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 3.13 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 C.7 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 3.4 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** $\epsilon = 4$

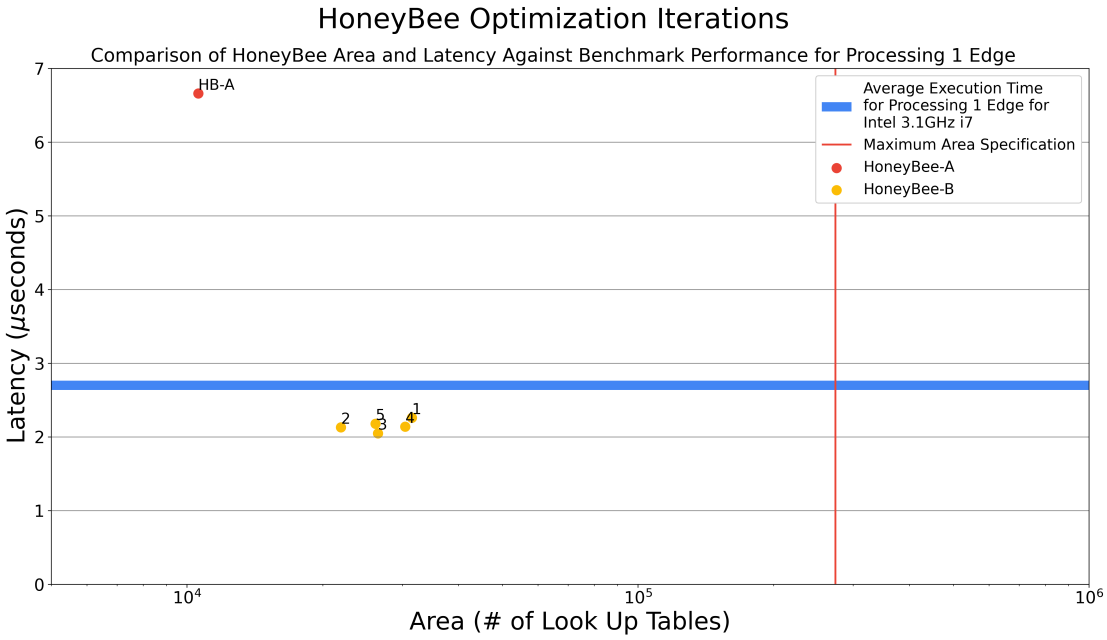
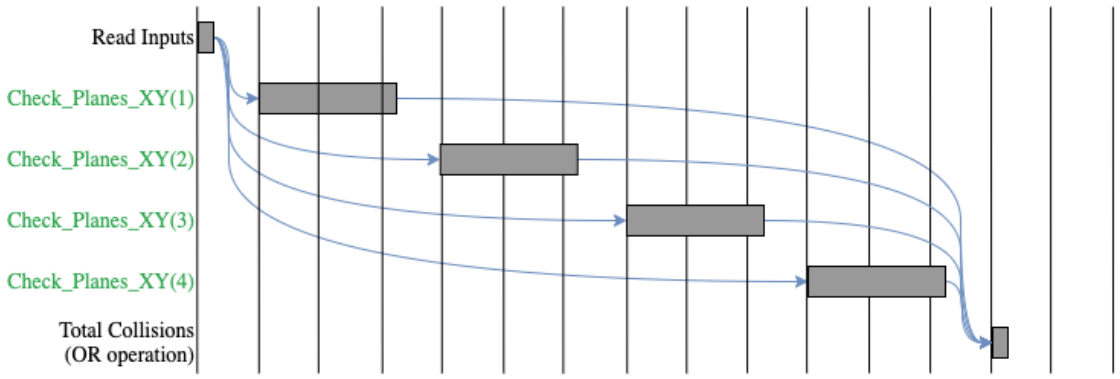


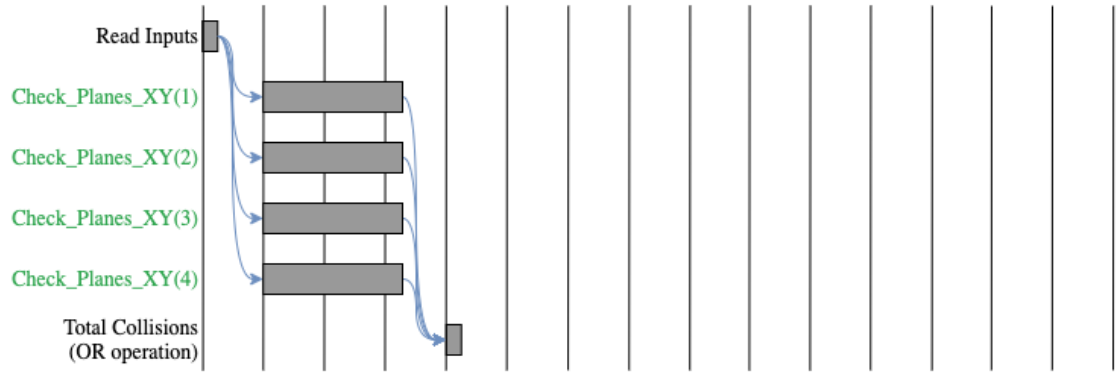
Figure 3.13: **HB-B Performance Against Benchmark CPU**
Variants of HB-B shown in Yellow.

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 3.14 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 3.14 shows how HB-B has only one instance of the module execute 4 times sequentially. HB-C, on the other hand, 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 3.5 and Figure 3.15.

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 $\epsilon = 4$**

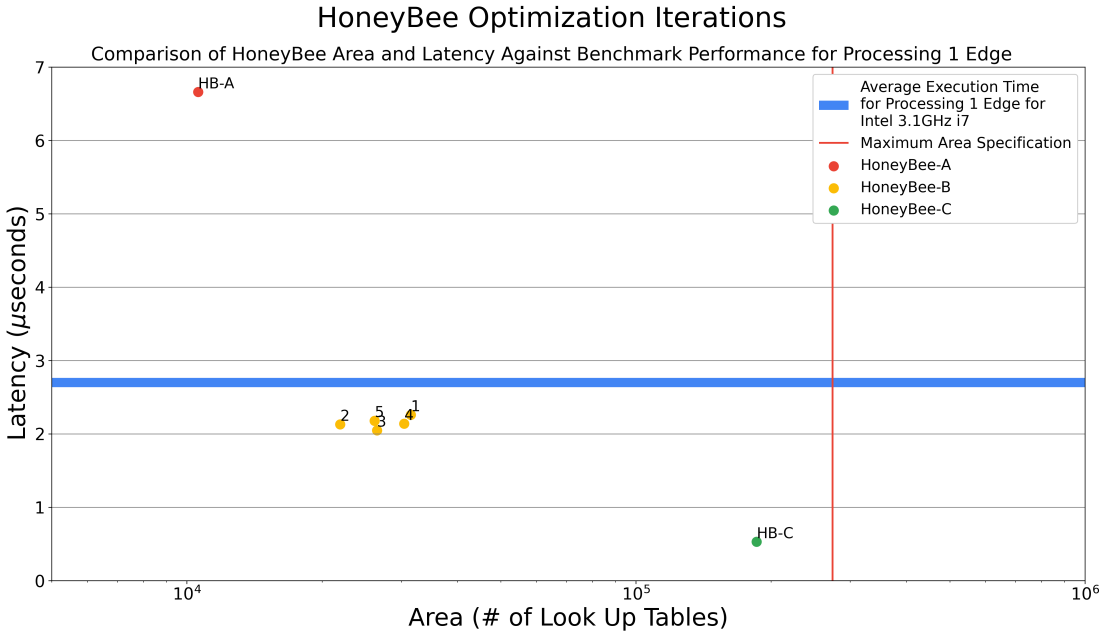


Figure 3.15: **HoneyBee-C Performance Against Benchmark CPU** HB-B variants shown in yellow, HB-C in green.

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 remaining 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

Recall the computer implementation hierarchy in Figure 1.1, made up of user-level applications, programming languages, the instruction set architecture, and the processor. Computer Architecture encompasses the two lower levels of this hierarchy. It comprises the design of the Instruction Set Architecture and Microarchitecture of a computer. Earlier, the Instruction Set was described as the "translator" between software and the processor. The **Instruction Set** is in reality a document defining the behaviour of a computer. The **Microarchitecture** is the implementation of the ISA; the physical computer processor that behaves in the way the ISAdefines.

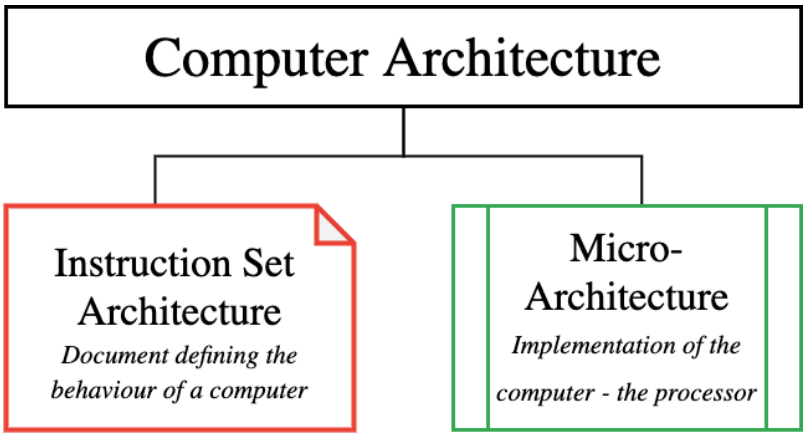


Figure 4.1: **Overview of the Field of Computer Architecture**

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 instructions that it can execute, along with the data types, memory model, and registers of the computer.
In more human terms, it can be thought of as a "contract" between hardware and software developers. It is a list of all the instructions that a computer will implement; all software will be compiled into these instructions, and all processors will be built to implement these instructions.

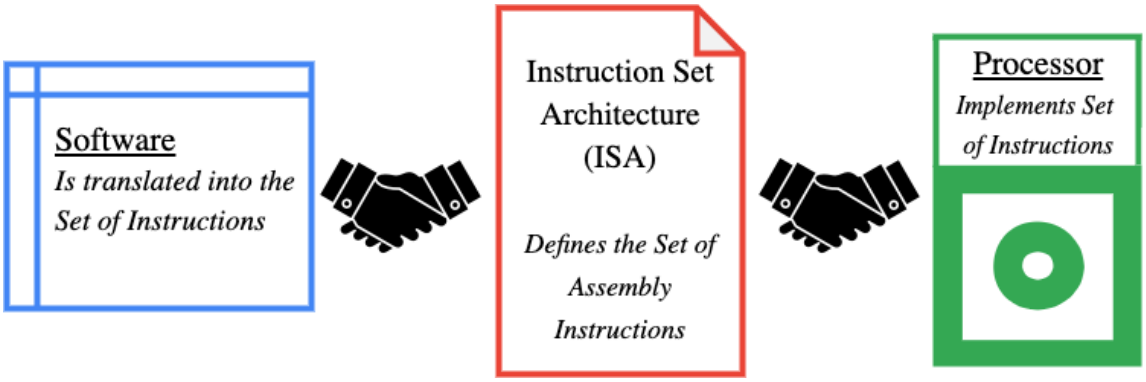


Figure 4.2: **The ISA is a Contract Between Software and Hardware Developers**

The two most important things that an ISA defines are a computer's **instructions** and **registers**. Instructions are the operations that the computer can execute. Registers can be thought of as slots in which the processor can store values, such as in Figure 4.3.

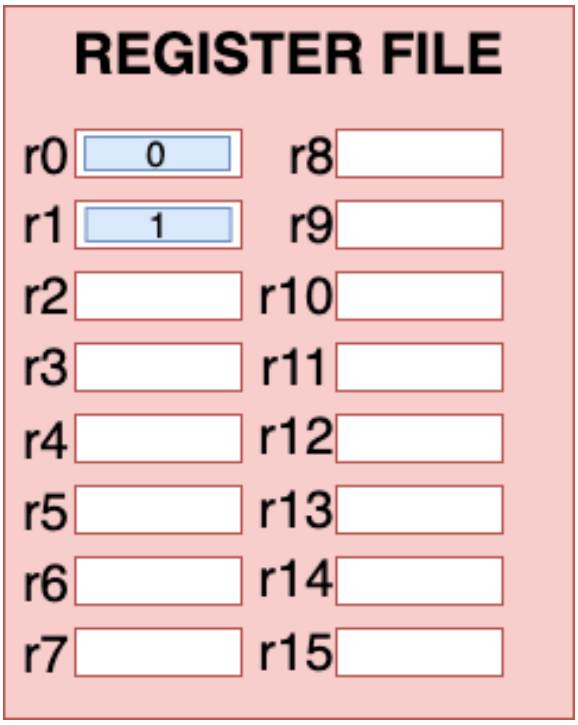


Figure 4.3: **Abstract Concept of a Register File** as a collection of slots in which a processor can store values.

Consider the RISC-V assembly instruction, `add`:

```
add rd, rs1, rs2
```

`rs1` and `rs2` stand for Source Register 1 and Source Register 2. `rd` stands for Destination Register. This instruction computes `rs1` + `rs2` and stores the result in `rd`. For example, consider an assembly program with the following three instructions and the register file from Figure 4.3 (note that r0 is initialized to the value of 0 and r1 to the value of 1.)

```
add r2, r1, r1
add r3, r1, r2
add r4, r2, r2
```

The first instruction computes `r1` + `r1` and stores the result in `r2`. Since `r1` is initialized to 1, this results in the value of 2 being stored in `r2`. The second instruction stores `r1` + `r2` in `r3`. Finally, the last instruction stores `r2` + `r2` in `r4`. When the program finishes

executing, registers 2, 3, and 4 have all been updated, as shown in Figure 4.4. I.e. new values have been stored in the computer's slots.

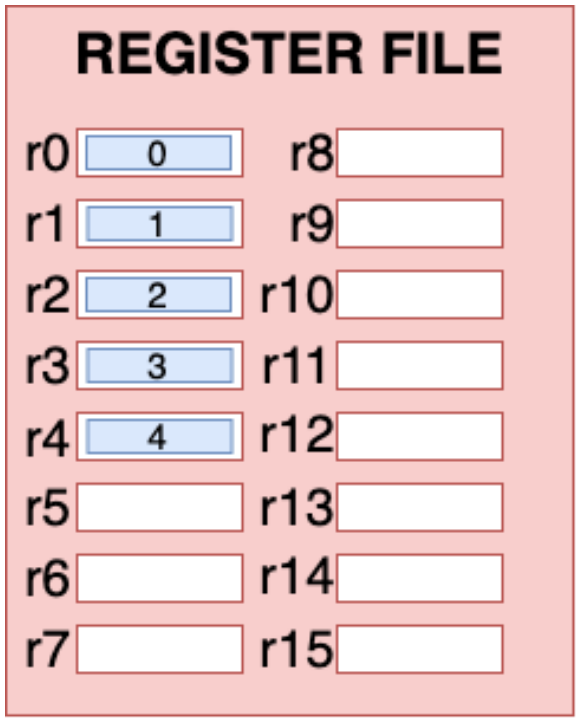


Figure 4.4: **Updated Abstract Register File** after having registers 2, 3, and 4 updated by an assembly program.

To recap, the above is an explanation of instructions and registers. The Instruction Set Architecture defines what registers a processor has and what instructions it can execute.

4.1.2 Microarchitecture

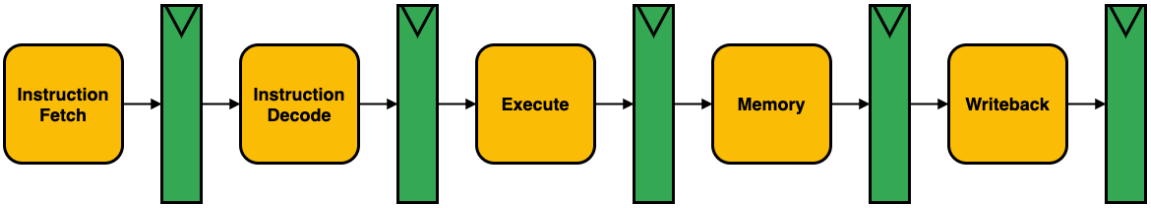


Figure 4.5: **5-Stage RISC Datapath**. Combinational logic stages are shown in yellow, and registers between stages are shown in green.

Microarchitecture refers to the actual implementation of the behaviour defined in the ISA. It is the design of the actual computer processor. Processors are typically implemented in stages, where each instruction goes through a certain number of stages to complete execution. Figure 4.5 shows the most simple layout of a 5-stage RISC Datapath (RISC is the computing principle on which RISC-V was founded. Appendix E.1 gives a background on RISC). 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. The values from memory or from the Execution stage are saved to one of the processor's registers in the **Writeback** stage.

4.2 RISC-V Instruction Set

4.2.1 RISC-V

RISC-V (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 following disadvantages of commercially popular ISAs.[2].

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

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 specifically optimized. This 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 non-standard extensions added to them. These base ISAs are not designed in such a way to overarchitect for a certain type of microarchitecure. **Standard extensions** are those that are generally useful and that are designed to not conflict with any other standard extensions. **Non-standard extensions** are those that may be highly specialized and may conflict with other standard or non-standard extensions. One way to think about it is that standard extensions would improve any computer and are endorsed by the RISC-V organization, whereas non-standard extensions are developed for very specific purposes, such as motion planning acceleration. This modularity and flexibility is part of what makes RISC-V such an attractive proposition for specialized computer architecture. Figure 4.6 demonstrates this modularity.

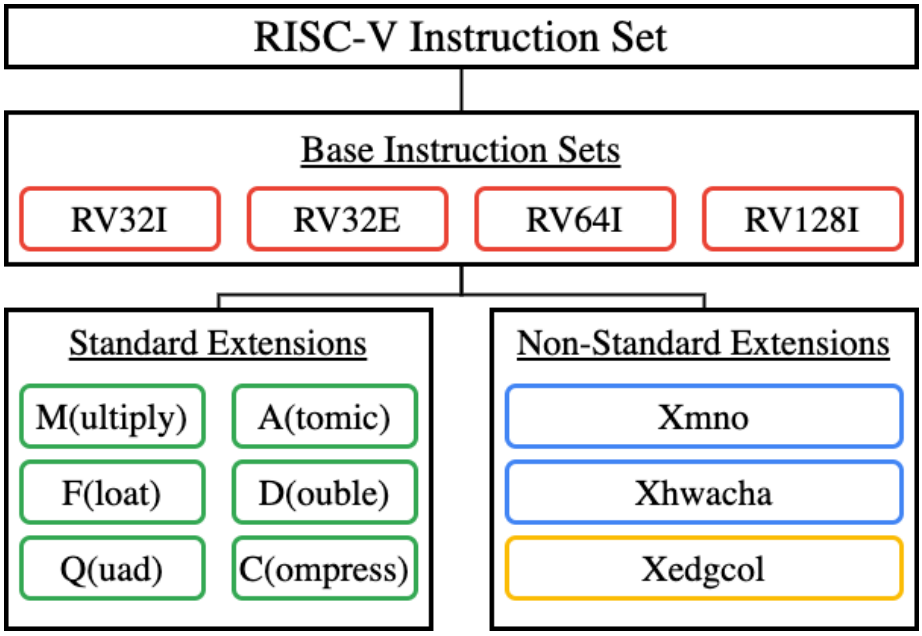


Figure 4.6: **RISC-V ISA Modularity**. It is clear that the RISC-V ISA is in fact a family of ISAs. Each is based on one of the "base" ISAs, which are each the smallest possible ISA to run just about any program. For more instructions and thus, better performance, any number of Standard or Non-Standard Extensions can be implemented on top of a base ISA.

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.

The following is an excerpt from the RISC-V Specification, outlining the RV32I base integer instruction set [3]:

> 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 …

RV32I was chosen as the base ISA for this project. The edge collision custom extension would be defined primarily extend the RV32I base ISA. Figure 4.7 is an updated system diagram of the overall project.

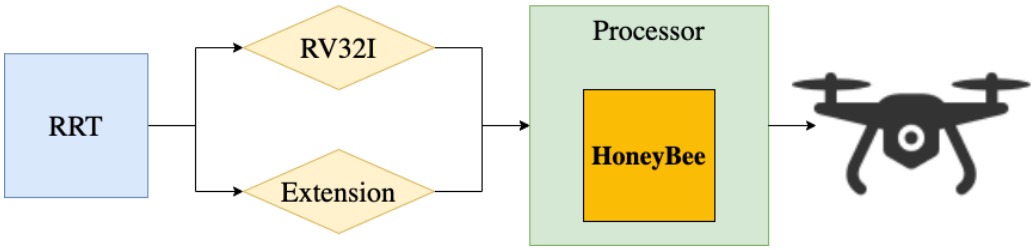


Figure 4.7: **Updated System Overview** showing how RRT is compiled through the RV32I ISA with the edge collision extension.

4.3 Defining a RISC-V Custom Extension

The goal of defining a custom extension in this context is to reduce the number of instructions necessary to compute collisions for a given edge. According to RISC-V convention, the extension is designated **"Xedgcol"** (X for non standard, edgcol as an abbreviation for its purpose). When it is implemented along with RV32I, the processor can be said to implement the RV32I_Xedgcol ISA.

4.3.1 Xedgcol Specifications

Table 4.1 outlines the general specifications for the Xedgcol extension. The specifications listed below reflect what was determined before the process of formally defining Xedgcol begun. As such, the specifications in the table are relatively vague. The rest of this section details the design decisions and the exact definition of each instruction and register.

Instructions	
Instruction	**Description**
Edge Collision	An instruction that commences edge collision detection computation. The information that is required is the *coordinates of the edge* and the *destination registers* for the result of the computation.
Load Edge	An instruction that loads the coordinate values that define the edge, $(x_0, y_0, z_0), (x_1, y_1, z_1)$, into the processor.
Registers	
Register	**Description**
Edge Registers	The 6 coordinate points need to be stored. Since the RV32I registers are only defined to hold integers and the coordinate values are floating point values, 6 new registers would need to be defined.

Table 4.1: **General Specifications for Xedcol RISC-V Extension**

Fixing Epsilon

Recall that the HoneyBee unit was parameterized for different values of ϵ, and that this would influence the number of bits in its output sequence. When defining an instruction, this value has to be constant. You may have also noticed that the results presented for HoneyBee in Chapter 3 were all for $\epsilon = 4$. This value was chosen because $4^3 = 64$. This can be stored in 2 32-bit registers. Table 4.2 lists the number of output bits required for each value of ϵ and how many 32-bit registers would be needed to store this result. No other values were suitable, so 4 was the obvious choice for the value of ϵ.

ϵ	**Bits**	**Registers**
1	1	1/32 bits
2	8	8/32 bits
4	64	2
6	216	6.75
8	512	16

Table 4.2: **Required Bits to Represent Output Collisions For Different Values of Epsilon**. ϵ values of 1 and 2 underutilise both the register space available and the benefits of more parallelization that would come from larger values. Values larger than 4 would use up far too many of the available registers. $\epsilon = 4$ completely utilizes only two registers.

4.3.2 Defining Xedgcol

Edge Collision Instruction

The following instruction was defined:

```
ECOL rd1, rd2
```

`ECOL` runs the edge-collision detection function and stores the result in two destination registers . The 32 Least Significant Bits (LSBs) are stored in `rd1` and the 32 Most Significant Bits (MSBs) in `rd2`. The above assembly instruction is (somewhat) human readable, but in the processor, every instruction is represented as a 32-bit sequence of binary values. Consider an empty 32-bit instruction, shown in Figure 4.8

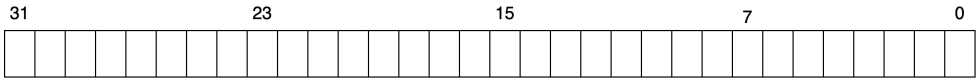


Figure 4.8: **Empty 32-Bit Instruction**. Reads left to right, Most Significant Bit (MSB) to Least Significant Bit (LSB)

The first piece of information to put in this instruction is the **opcode**. An opcode (short for operation code) is the instructions identity. It tells the processor which operation to execute. The opcode for the `ecol` instruction was defined as the 3-bit sequence `100`. The opcode is stored in the 3 LSBs of the instruction, as shown in Figure 4.9

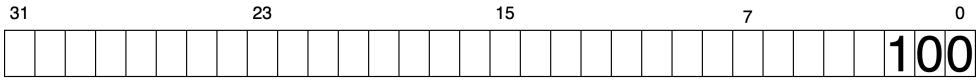


Figure 4.9: **ECOL Instruction with Opcode**

Next, the two **destination registers** must be added to the instruction. These are the regular RV32I registers. They are addressable with 5-bit values, shown in Figure 4.10.

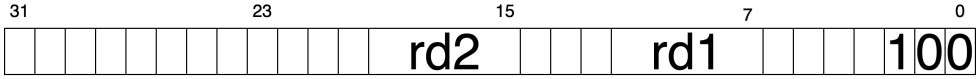


Figure 4.10: **ECOL Instruction with Opcode and Destination Registers**. The position of two destination register addresses is such that it minimized the extra logic and instruction decoding necessary to implement this instruction.

The `ECOL` instruction specifies two destination registers, but no source registers. In other words, how does the processor know the coordinates of the edge? These coordinates are stored in the newly defined edge registers.

Edge Registers

Consider, an edge must be represented by 6 32-bit floating-point (i.e. fractional) numbers. This information, 192 bits in total, cannot all fit in a single 32-bit instruction. They also can't be stored in existing RV32I registers, which are defined to only store integers. Storing floating-point numbers in integer registers, while theoretically possible, is extremely bad practice. To store the floating point coordinates in registers, a new register set would need to be defined.

6 floating point registers were defined for the Xedgcol extension. These registers are specifically for the purpose of holding the coordinates of the edge for which intersections will be calculated. Table 4.3 lists these registers.

Register	ABI Name	Description
e0	px0	X coordinate of edge's first point
e1	py0	Y coordinate of edge's first point
e2	pz0	Z coordinate of edge's first point
e3	px1	X coordinate of edge's second point
e4	py1	Y coordinate of edge's second point
e5	pz1	Z coordinate of edge's second point

Table 4.3: **Edge Coordinate Registers** as defined in the Xedgcol ISA extension

Load Immediate Edge Instruction

Since Xedgcol was designed to be implemented alongside only RV32I (which, as stated, does not support floating-point binary representation), the Load Edge Immediate instruction was defined to load an immediate float value into one of the 6 edge registers.

```
LI.e rd, imm
```

The `LI.e` instruction takes a destination register and an immediate floating-point value. It stores the immediate value `imm` in the destination register `rd`. To build this instruction, an opcode (`000`) and a destination register is required. Xedgcol has an opcode width of 3, and since there are only 6 edge registers, they are addressable by a 3-bit address. This leaves 26 bits remaining in the instruction for the immediate.

A floating point immediate is 32 bits, so the 6 LSBs are cut-off (which does result in some loss of precision, but only slightly) in the instruction.

The format for the LI.e is shown in Figure 4.11

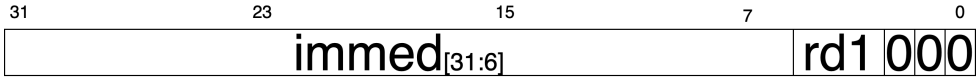


Figure 4.11: **Load Immediate Edge Instruction Format**

The formal Xedgcol definition decoument can be found in Appendix D. The extension has been documented following RISC-V conventions.

4.4 PhilosophyV

Philosophy 4 was written in 1903 by Mr. Owen Wister of the Class of 1882 (author of *The Virginian* and the founder of the Western literary genre). It 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 ill-prepared. 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 hopelessly ill-prepared. As such, this processor was named **PhilosophyV**; both in reference to the RISC-V ISA for which it was designed, and to the process of its implementation, which at times seemed very much like a sequel to Mr. Wister's novel.

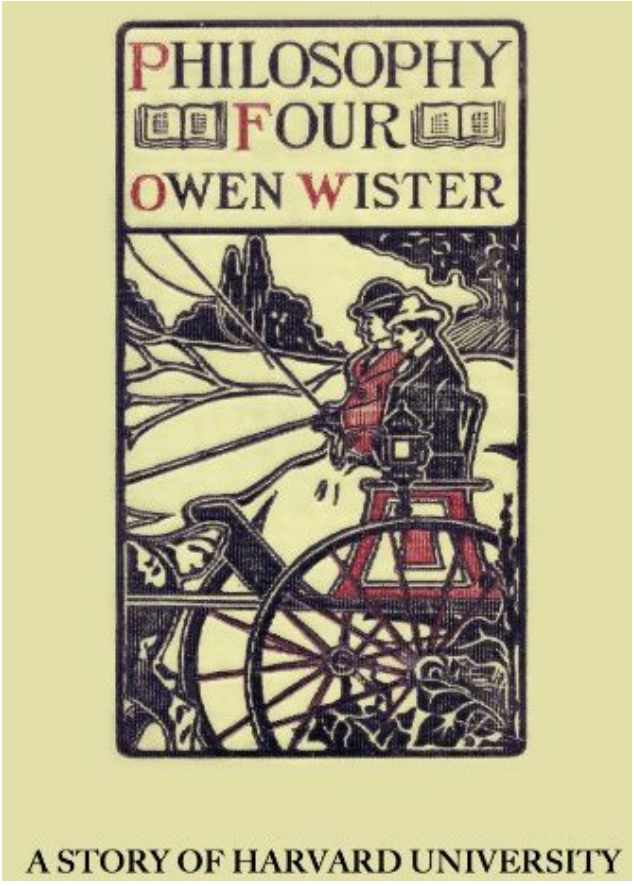


Figure 4.12: **Cover of Philosophy 4**. Source: Houghton Library, Harvard University

The purpose of implementing a functional RISC-V processor was to verify that the design of the Xedgcol extension was viable. Initially, the hope was to find an existing open-source implementation, of which there are many, that I could build on. A significant period of time was spent trying to become familiar with the Rocket Core[12], a large open-source RISC-V implementation. However, the project was so sophisticated that learning its infrastructure and the neccesary tool-chains became a massive project in itself. This was the case for many open-source projects. Those that lacked such sohpisticated code bases also lacked proper documentation and were not verified to be correct. As a result, it ended up being faster and simpler to implement a lightweight RISC-V processor from scratch.

4.4.1 RV32I Implementation

The first step was implementing a processor that implemented the RV32I Base ISA. Figure 4.13 shows a highly simplified schematic of the RV32I PhilosophyV Core. Appendix E.2 provides a detailed schematic.

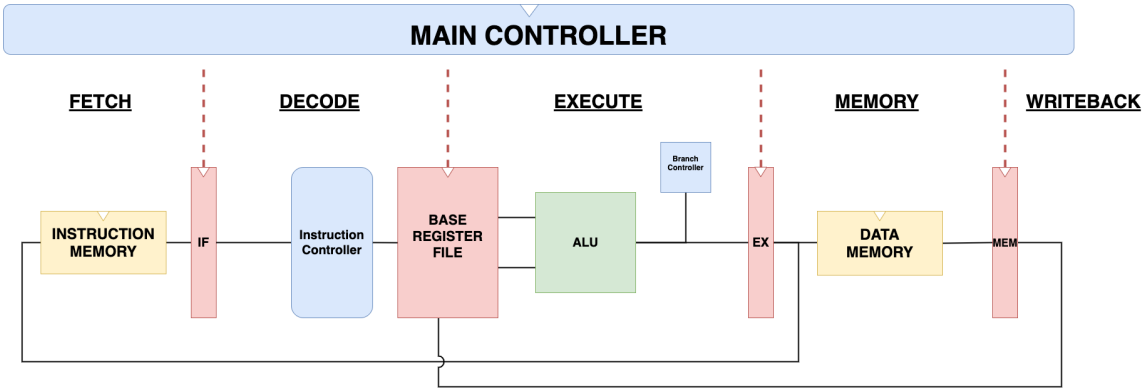


Figure 4.13: **Simplified Schematic of the RV32I PhilosophyV Core**

The design was that of a simple 5-stage non-pipelined processor with no branch-prediction or other optimizations. In other words, it's slow. However, speed was somewhat irrelevant in this implementation, as its only purpose is to prove the validity of the Xedgcol ISA extension (implementing a processor with comparable performance to an Intel i7 CPU, for example, would be well beyond the scope of this thesis).

PhilosophyV was implemented in Verilog (an HDL). Simulations were carried out in Vivado Design Suite. The processor's correctness was verified at the module level and at the core level. Module testing was fulfilled by Verilog test benches that checked the functional correctness of each module (e.g. the ALU). The overall core was tested by running different

complex assembly programs through the processor. At the end of the program's execution, the state of its register file was compared to its expected state.

4.4.2 RV32I_Xedgcol Implementation

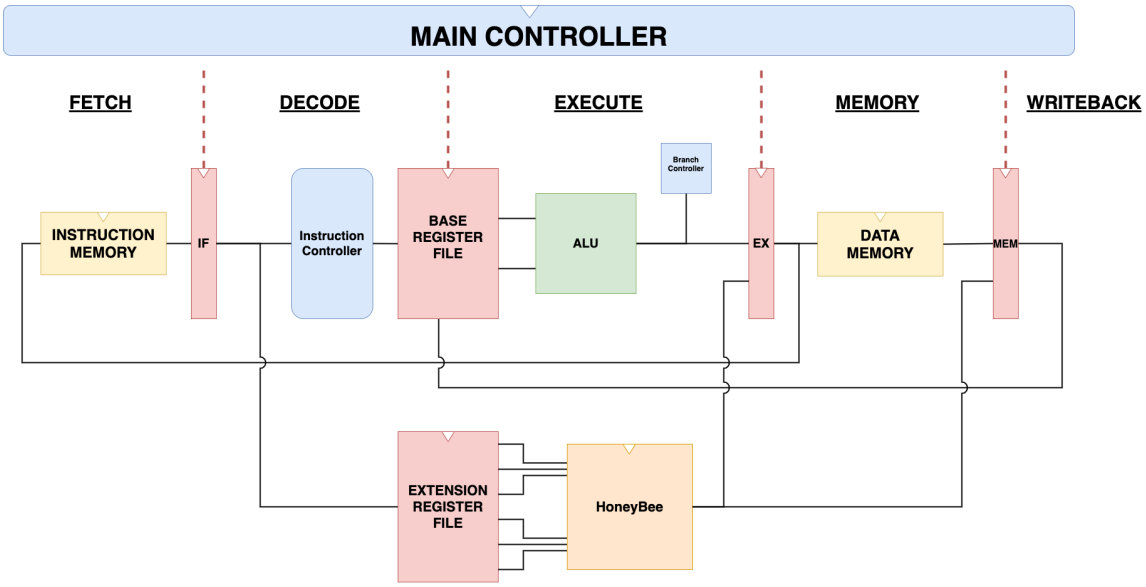
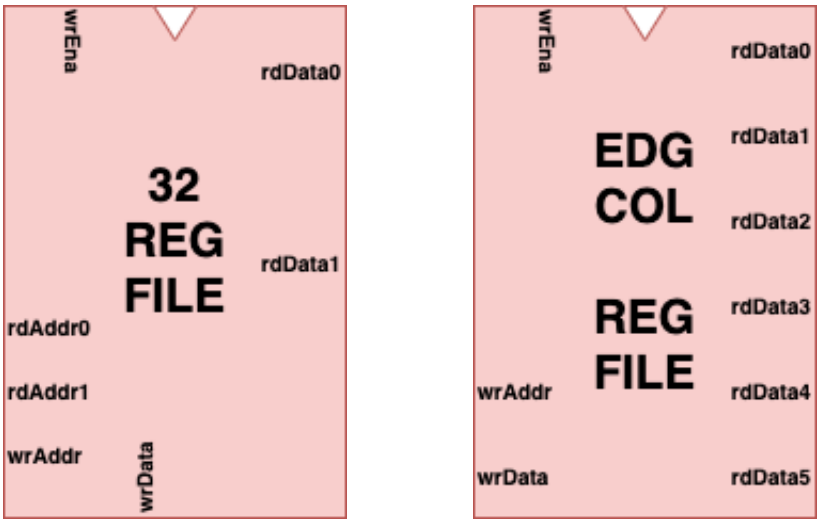


Figure 4.14: **Simplified Schematic of the RV32I_Xedgcol PhilosophyV Core**

Implementing the Xedgcol extension was relatively simple. First, a **new register file** had to be implemented to support the new registers `e0-e5`. The register file was implemented slightly differently to normal; It was implemented with a write address port, a write data port, and then 6 data read ports that always output the value of the 6 registers. This is shown in Figure 4.15.



(a) (b)

Figure 4.15: **PhilosophyV Register Files**. The 32RVI register file contains 32 registers. The ports `readData0` and `readData1` output the data held at the address provided to `rdAddr0` and `rdAddr1` respectively. The Xedgcol register file is able to constantly output all 6 values held within it. Since the Xedgcol ISA only defines their use for one instruction, these ports can be wired directly to the HoneyBee unit.

The `LI.e` instruction writes to this register file. In the instruction, the float immediate is only in fact the 26 most significant bits, as room must be left in the instruction for the 3 bit register address and the 3 bit opcode. As such, this float is extended with zeros in the least significant bit range before being wired to the wrData port of the register file.

Secondly, to implement the `ecol` instruction, the HoneyBee unit was added. Its 6 input ports, defining the coordinates of the edge, were wired directly to the output ports of the Xedgcol register file. Its 64 output bits were split to be written to the two destination registers. The 32 MSBs are written to rd2 in the memory stage, and the 32 LSBs are written to rd1 in the writeback stage. Its control interface was wired to the main controller and the main controller's logic was updated. In this unoptimised processor design (to keep things simple) the processor stalls while it waits for the HoneyBee unit to complete execution.

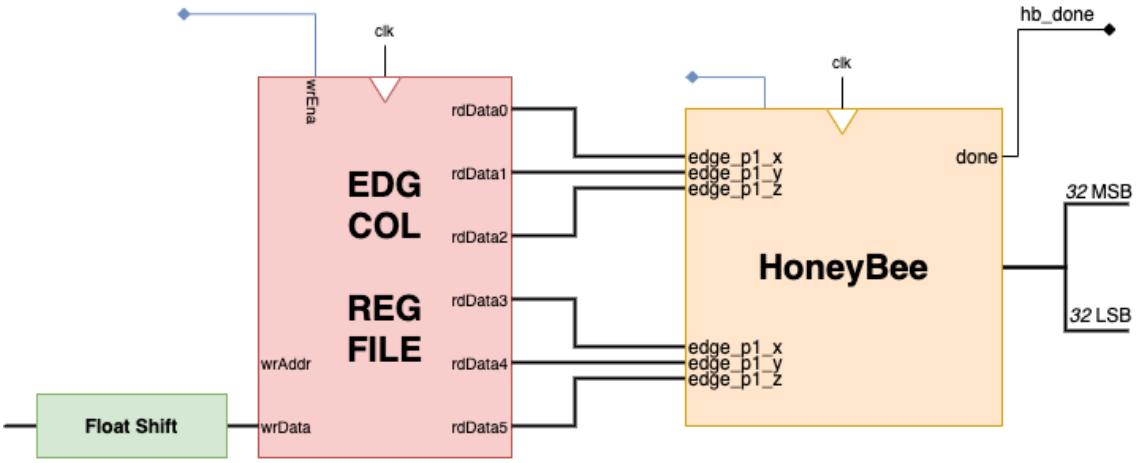


Figure 4.16: **Implementation of HoneyBee in RV32I_Xedgcol PhilosophyV**. The data written to the register file is extended with zeros. The values of each of the 6 Xedgcol register are wired directly to the 6 input ports of HoneyBee. HoneyBee's output is written to two destination registers.

4.4.3 Verification

With the PhilosphyV core implementing the RV32I_Xedgcol ISA, assembly tests were written to verify the viability of the Xedgcol extension. Consider an edge that is defined by the points $(0.5, 0.75, 0.25)$ and $(1.75, 1.25, 1.5)$. The assembly instructions to execute the edge collision functionality are as follows:

```
# Load immediate coordinate values
LI.e px0 0.5
LI.e py0 0.75
LI.e pz0 0.25
LI.e px1 1.75
LI.e py1 1.25
LI.e pz1 1.5
# Execute edge collision function
ECOL x6, x7
```

The collision bit sequence stored in registers `x6` and `x7` can be compared against an occupancy grid map. Over multiple tests, the correct collision bits were stored in the correct registers. It was concluded from these tests that the Xedgcol extension was a viable solution.

Chapter 5

Conclusion

5.1 Summary of Results

Motion Planning in Software

The Rapidly-exploring Random Tree algorithm was successfully implemented in C. Rigourous experimentation was conducted to determine the optimal set of parameter values for quickly finding a collision-free path with high probability for a given map size. Once these optimal parameters were determined for different map sizes, thousands of simulations were run to find the bottleneck function of RRT. It was found that the edge collision detection function consumed as much as 70% of CPU execution time in 3D.

Motion Planning in Hardware

The HoneyBee unit was designed to implement the computation of edge intersections. High levels of hardware parallelization were utilized to reduce the latency of computing one edge from 6.66 μseconds to 0.53 μseconds. This translated to an improved throughput of 1,886,792 edges/second from (150,150 edges/second). Moreover, this was achieved in an acceptable area-on-FPGA.

Motion Planning Architecture

The Xedgcol non-standard RISC-V extension was defined. This ISA extension introduces two new instructions (`ECOL` and `LI.e`), along 6 new floating-point registers `e0-e5`. This extension reduces the number of instructions required to compute collisions for one edge from tens of thousands to only one (if you include the 6 LI.e calls to load the coordinate, then it requires seven). Philosophy V, a RISC-V processor, was implemented from the ground up to verify the extension. HoneyBee was implemented into PhilosophyV and tests of the Xedgcol extension conducted. The extension performed as defined.

5.2 Evaluation of Success

This thesis had four objectives:

Project Objectives

1. Determine the computational bottleneck of a commonly used motion planning algorithm.
2. Implement a functional hardware module to replace the bottleneck function.
3. Define a motion planning extension to the RISC-V ISA
4. Build a fully functional motion-planning processor that implements the extended RISC-V ISA

The first objective was achieved. RRT was implemented from scratch, which may cast doubt as to how accurately its performance reflects real world implementations. The edge collision function was in fact implemented very realistically. The UAV was represented as a 3D rectangular prisim and edge collision detection was based on solid geometric principles. If anything, more time was spent making sure that the edge collision function was as fast as possible in software, rather than on the other 4 key functions. The function that finds the nearest node in the graph was the second most computationally intense. There are many proven algorithmic optimizations for this function that were not applied in this implementation. Therefore, any uncertainty in the results of RRT analysis would in fact *under-represent* the computational load of edge collision detection.

The second objective was also succesfuly completed. HoneyBee acheived a reduction in latency that was greater than its specification. HB-C could compute edge collisions 5 times faster than a professionally developed multi-core processor from Intel. Admittedly, once this performance was achieved, no further development was attempted. There still exists the opportunity for slight area optimizations and potentially even greater reductions in latency, though they would most likely be marginal.

Of most significance to the overall goal of this thesis is the third objective. Successfully defining a RISC-V extension is the most compelling evidence for the future application of RISC-V in developing motion planning specific processors.

The Xedgcol extension is simple and effective. This thesis took the approach of defining an instruction that would execute an entire function of RRT (edge collision). Since edge collision detection is required in most motion planning algorithms, the Xedgcol extension is widely applicable in the space. Another benefit of the Xedgcol design is how easy it is to implement in microarchitecture. The instructions were formatted in a way that reduced the the extra logic required for instruction decoding. Furthermore, designing a hardware unit like HoneyBee, while somewhat challenging, is made quite easy with tools such as HLS.
To cite some disadvantages of its design, it is perhaps not very elegant. It condenses an entire function into one instruction, which, while simple, does not afford an assembly programmer much flexibility. It also "overarchitects" for a certain implementation. There isn't really any other way to implement this particular extension other than implementing a hardware unit similar to HoneyBee (this does show, however, just how specific custom extensions can be).
That being said, it does serve its purpose. It follows RISC-V convention and it correctly, simply, and effectively defines instruction set architecture for motion planning purposes.

The fourth objective was to desiging a RISC-V motion plannning processor. It was alluded to in Chapter 4 that this was not a fast processor. This is true; however, it was designed with simplicity in mind, for the purpose of verification, not performance. It correctly implements the RV32I_Xedgcol instruction set, and verifies that the Xedgcol extension is compatible with the RV32I base intruction set. For the scope of this thesis, this was a success.

5.3 Future Work

The overall mission of this thesis was:

Mission
To present a proof-of-concept for extending and implementing RISC-V to develop motion planning specific processors.

The driving motivation for this mission is that RISC-V has not yet been embraced by computer architects looking to improve computation of motion planning. Autonomous robots, and in particular, autonomous UAVs, have the potential to change the world in which we live. Imagine warehouses filled with swarms of autonomous UAVs, picking and packing orders without a human in sight. Imagine operations being carried out in dangerous, complex environments by drones with full autonomy. For this to be realised, however, motion planning specific processors need to be developed. Whereas commercial ISAs present many

barriers for doing this; RISC-V is all but designed for it. Suprisingly though, at the time of writing, there has been no published work on the application of RISC-V for developing motion planning specific processors. It is hoped that this thesis may encourage efforts to be made in this space, perhaps in one of the following areas.

Edge Collision Hardware

As mentioned above, the optimization of HoneyBee stopped once performance specifications for this project were met. It is feasible that there is a non-trivial amount of latency reduction still available with further optimization. Alternatively, perhaps there are other approaches to accelerating edge collision in hardware. Even if not within the RISC-V ecosystem, achieving faster motion planning will at some level depend on reducing the computational load of edge collision detection.

Production Quality RV32I_Xedgcol Processor

It was well beyond the scope of this thesis to implement a production quality processor. It would be an interesting experiment to run comparison tests of RRT execution time between a general purpose Intel CPU and a professionally developed, multi-core RISC-V processor that implements the Xedgcol extension.

Better Motion Planning Extensions

The Xedgcol extension has some drawbacks that were highlighted above. As a proof of concept, it was very successful, but the biggest opportunity for future work lies in the development of better motion planning extensions. As more computer architects work on these extensions and better extensions are defined, the more of a developer following they get. This developer community builds, shares, and improves the extension and the toolchains for implementing that extension. For example, there is no compiler support for the Xedgcol extension - using it has to be done manually in assembly code, rather than high level languages. Compiler support may be implemented for a very popular non-standard extension. Of all potential future work to come from this thesis, it is hoped that the active development of RISC-V motion planning extensions occurs.

To conclude with a call to arms: This thesis provides a proof-of-concept for developing application specific processors for RISC-V with relative ease. Instruction Set Architecture is the most important interface of a computer, and RISC-V is the first commerically viable opportunity to open up this interface to the wider developer community. It is not an opporunity to be neglected. If this humble prototype can be a convincing argument for this, then it will be considered a success by its author.

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

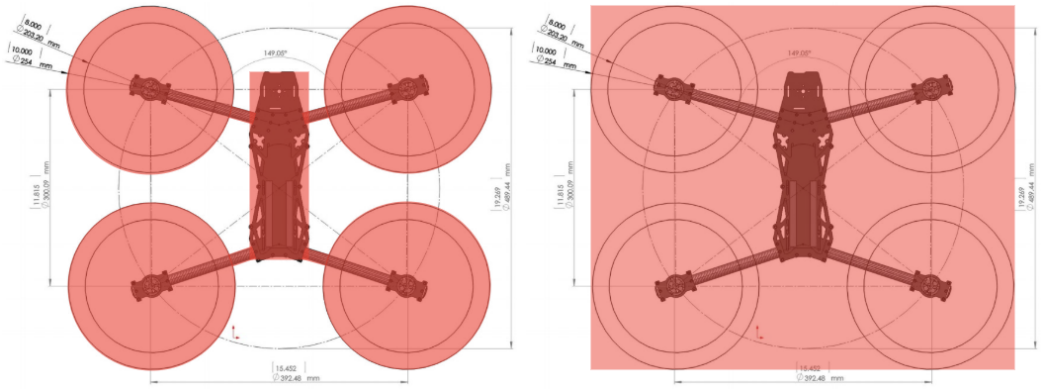
Appendix B

RRT Supporting Documentation

B.1 Justification of Modelling UAV as Prism

While it is possible for a UAV to be modelled in precise detail, taking into account its exact shape, more often UAVs are modelled as a 3D prism in motion planning problems, for the following reasons:

- It is a rare case that the negative space gained by modelling in such detail is utilised
- Representation of the drone's configuration is much more complex.
- Computing edge collisions is much more computationally intensive.



(a) (b)

Figure B.1: **Modelling a UAV as a Rectangular Prism**. Red highlights demonstrate the configuration model, overlayed over the exact schematic. Figure B.1a shows how a drone can be modelled in high detail, but gains little useful free space when compared with Figure B.1b, which models a drone as a rectangular prism.[13]

B.2 Full Technical Specifications for RRT Implementation

General Specifications	
Requirement	**Description and Justification**
Implemented in C/C++	As outlined in Section 1.2.2, 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, unlike higher-level languages such as Python.
3D Workspace	The computational requirements of RRT in 3D differs somewhat to that in 2D. Since autonomous UAVs operate in 3D space, it was neccesary to have a 3D implementation to analyse.
UAV modelled in 3D as a rectangular prism	In theory, it is possible to model a UAV much more precisely than a rectangular prism, taking into account its shape and negative space. 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 efficient). See Appendix B.1 for justification of this.
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.

Table B.1: **General Technical Specifications for RRT Implementation**

Required Parameters	
Parameter	**Description and Justification**
ϵ (a.k.a. Δq)	The maximum difference between two configurations. Larger values of ϵ can solve less obstacle dense problems faster, but take longer to solve problems with tight corners.
K	The maximum number of configurations. This is largely correlative to the amount of time the user will allow the algorithm to run. Larger values of K will take longer but generate better paths, while smaller values will execute for less time but generate more jagged paths or may not reach the goal node. The value of K was varied to find the minimum execution time while still reaching the goal with high probability.
DIM	The upper bound of each axis of a $DIM \times DIM \times DIM$ Workspace. Larger values leave more space to be explored, and thus require larger values of K to reach the goal with high likelihood.
Goal Bias	The given probability that the graph will extend the graph ϵ distance from an existing configuration to a new configuration in the direction of the goal.

Table B.2: **Required Parameters for RRT Implementation**

B.3 Assessment of Existing RRT Implementations

	General Requirements				**Parameters**			
Repository	Language	2D/3D	Object Model	Collision Detect	ϵ	K	DIM	Goal Bias
RoboJackets[14]	C++	2D	Point	Complete	Yes	Yes	Yes	No
Motion-Planning[15]	Python	ND	Point	Incomplete	Yes	Yes	No	Yes
Sourishg[16]	C++	2D	Point	Incomplete	Yes	Yes	No	No
Vss2sn[17]	C++	2D	Point	Complete	Yes	Yes	No	No
olzhas[18]	Matlab	2D	Point	Complete	Yes	Yes	No	No

Table B.3: **Evaluation of Existing Open-Source Implementations of RRT**. Links to Github repositories can be found in the Bibliography.

B.4 Implementation of Key RRT Functions

Algorithm B.1: `getRandomConfig()` as implemented for RRT

Inputs: Dimensionality N,
Upper Axis Bound DIM
Output: Random Configuration q

$q.x \leftarrow$ randomFloat(DIM)
$q.y \leftarrow$ randomFloat(DIM)
$q.\alpha \leftarrow$ randomFloat(2π)
if $N == 3$ **then**
  $q.z \leftarrow$ randomFloat(DIM)
  $q.\beta \leftarrow$ randomFloat(2π)
  $q.\gamma \leftarrow$ randomFloat(2π)
end
return q;

Where `randomFloat(max)` returns a float between 0 and `max`.

Algorithm B.2: `findNearestConfig()` as implemented for RRT

Inputs: Graph G,
New Configuration q_{new}
Output: Nearest Configuration $q_{nearest}$

$q_{nearest} \leftarrow G.q_{init}$
for $k = 0$ to G.existing_nodes **do**
  if distance(q_{new}, $G.q[k]$) < distance(q_{new}, $q_{nearest}$) **then**
    $q_{nearest} \leftarrow G.q[k]$
  end
end
return $q_{nearest}$

Where `distance(`q_1`,` q_2`)` returns the Euclidean distance between two configurations.

Algorithm B.3: `stepFromNearest()` as implemented for RRT

Inputs: Configuration in Graph $q_{nearest}$,
New Configuration q_{new},
Goal Bias B,
Maximum Step Distance ϵ,
Graph G
Output: Updated New Configuration q_{new}

if distance($q_{nearest}$, q_{new}) $> \epsilon$ **then**
  if randomFloat(1) $< B$ **then**
    $q_{new} \leftarrow$ stepTowardConfig($q_{nearest}$, $G.q_{goal}$)
  end
  else
    $q_{new} \leftarrow$ stepTowardConfig($q_{nearest}$, q_{new})
  end
end
return q_{new};

Where `stepTowardConfig(`q_1`,` q_2`)` returns a configuration ϵ from q_1 in the direction of q_2.

Algorithm B.4: `configCollision()` as implemented for RRT

Inputs: Dimensionality N,
Occupancy Grid Map (N-Dimensional Array) O,
Configuration q
Output: Boolean

if $N == 2$ **then**
  return O[gridLookup($q.x$)][gridLookup($q.y$)]
end
else
  return O[gridLookup($q.x$)][gridLookup($q.y$)][gridLookup($q.z$)]
end

Where O is a N-Dimensional array of booleans, with True representing an occupied grid and false representing an unoccupied one. `gridLookup()` is a function that maps a floating point coordinate to the correct integer of the grid in which it resides. For a map resolution of one, this is as simple as rounding a float down to an integer.

While seemingly complex, the above algorithm merely steps through the mathematical process of checking the relevant x, y, and z planes for a point of intersection with the edge e. It then looks up the OGM O to see if the grid corresponding with the point of

Algorithm B.5: `configCollision()` as implemented for RRT for 3D

Inputs: Edge e,
Occupancy Grid Map (3-Dimensional Array) O,
Maximum Step Distance ϵ
Output: Boolean

$q_{min} \leftarrow$ minConfig($e.q_1$, $e.q_2$)
for ($x = q_{min}.x$ to $q_{min}.x + \epsilon$) **do**
  $q_{intersection} \leftarrow =$ edgeIntersectsPlane(e, x)
  if $O[q_{intersection}.x][q_{intersection}.y][q_{intersection}.z]$ **then**
    return true
  end
end
for ($y = q_{min}.y$ to $q_{min}.y + \epsilon$) **do**
  $q_{intersection} \leftarrow =$ edgeIntersectsPlane(e, y)
  if $O[q_{intersection}.x][q_{intersection}.y][q_{intersection}.z]$ **then**
    return true
  end
end
for ($z = q_{min}.z$ to $q_{min}.z + \epsilon$) **do**
  $q_{intersection} \leftarrow =$ edgeIntersectsPlane(e, z)
  if $O[q_{intersection}.x][q_{intersection}.y][q_{intersection}.z]$ **then**
    return true
  end
end
return false

intersection is occupied. If so, then it reports a collision by returning True. The function `edgeIntersectsPlane` follows the geometrical process of detecting a segment-plane intersection outlined in Appendix B.5. q_{min} is calculated to be the origin point of the grid closest to the origin. In other words, the algorithm does not check for intersections throughout the entire map, only the maximum number of grids that could possible be intersected by the edge e, given the location of the two points of the edge, $e.p_1$ and $e.p_2$, and the maximum edge length ϵ. The algorithm for `edgeCollision()` in 2D can be inferred from the above, checking segment-line intersections for x and y lines.

B.5 Geometrically Determining Segment-Plane Intersection

The method of edge collision detection in this project's implementation of RRT relies on detecting segment-plane intersections. The planes are always set up to be parallel with either the xy plane, the xz plane, or the yz plane. Figure B.2 demonstrates this point.

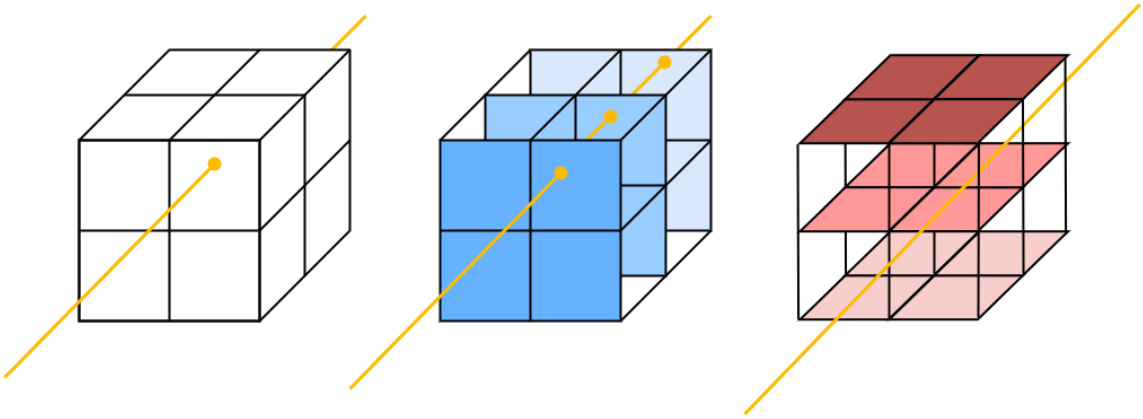


Figure B.2: Using Parallel Planes to determine Edge Collisions with Grids

A plane can be defined by 3 points, P_a, P_b, and P_c. In practice, the points defining a plane parallel to the xy plane would have the following points:

$$P_a = (x, y, z)$$
$$P_b = (x + \Delta x, y, z)$$
$$P_c = (x, y + \Delta y, z)$$

Two vectors, $\vec{AB}$ and $\vec{AC}$ can be determined.
The normal to the plane is the cross product:

$$\vec{AB} \times \vec{AC}$$

And the equation of the plane written as:

$$a(x - x_0) + b(y - y_0) + c(z - z_0) = 0$$
$$ax + by + cz = ax_0 + by_0 + cz_0$$

Where $< a, b, c >$ is the normal to the plane and (x_0, y_0, z_0) is one of the points P_a, P_b, or P_c. The RHS can be set to equal d, leaving:

$$ax + by + cz = d$$

Now, the equation of a line can be written in the form:

$$ax + by + cz = 0$$

And can be parameterized in the following form:

$$\begin{cases} x = & x_1 + t(x_2 - x_1) \\ y = & y_1 + t(y_2 - y_1) \\ z = & z_1 + t(z_2 - z_1) \end{cases}$$

To find the point of intersection, we substitute the equation of the line into the equation of the plane, yielding:

$$a(x_1 + t(x_2 - x_1)) + b(x_1 + t(x_2 - x_1)) + c(x_1 + t(x_2 - x_1)) = d$$

Rearranging to find an expression for t:

$$t = \frac{d - (ax_1 + by_1 + cz_1)}{a(x_2 - x_1) + b(y_2 - y_1) + c(z_2 - z_1)}$$

Knowing t, we can find the point of intersection, P_X to be:

$$\begin{cases} x_X(t) = & x_1 + t(x_2 - x_1) \\ y_X(t) = & y_1 + t(y_2 - y_1) \\ z_X(t) = & z_1 + t(z_2 - z_1) \end{cases}$$

Finally, the following equalities are evaluated to see if the point lies on the segment:

$$x_1 \leq x_X \leq x_2$$

$$y_1 \leq y_X \leq y_2$$

$$z_1 \leq z_X \leq z_2$$

If so, then the grids corresponding to the point of intersection can be marked as intersected.

B.6 Timing Methodology of RRT Analysis

VTune Profiler

VTune Profiler is an application for software performance analysis. It provides functionality to examine hot-spots for CPU execution time through a top down analysis. The top down analysis tool shows the percentage of CPU time taken up by each function. It was used to initially profile the algorithm's performance.

Internal Timing

There are several limitations to using VTune Profiler. First, it can only profile software running on Intel processors, which implement the x86 ISA. In anticipation of potentially needing to run performance analysis on a RISC-V processor, another method was required. Secondly, VTune Profiler takes a long time to run, as it needs to conduct a lot of analysis that is extraneous to the purpose of this thesis. This became prohibitive when it came to conducting hundreds of tests for different parameterizations, with each test running RRT a minimum of 100 times. Finally, it was not customizable to ignore certain parts of the implementation, such as logging functionality. While the implementation was designed in such a way that these should not intefere, it led to a lot of irrelevant data. A simple and effective alternative for measuring execution performance was to insert timing functionality into the software itself.

Internal timing was implemented based on the inbuilt C `clock()` function and `CLOCKS_PER_CYCLE` macro, and wrapping each function of interest in a performance tracking struct. This can be seen in the project's RRT sub-repository under `performance.h`.

Comparison

Before proceeding to use the internal timing method, it was important to verify that this method yielded similar results to VTune Profiler for the same program. Table B.4 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. For good measure, a χ^2 test of hypothesis was conducted and for one degree of freedom showed more that acceptable results.

function	Vtune Profiler		Internal Timing		χ^2
	time (s)	time (% total)	time (s)	time (% total)	
A	0.488	57.4%	0.497	57.6%	0.00016
B	0.2	23.5%	0.198	23.1%	0.00002
C	0.102	12.0%	0.099	11.5%	0.00009
D	0.048	5.7%	0.049	5.6%	0.00002
E	0.012	1.4%	0.019	2.2%	0.00408

Table B.4: Comparison of Timing Methods

B.7 Execution Time of 2D and 3D RRT for Different Map Sizes

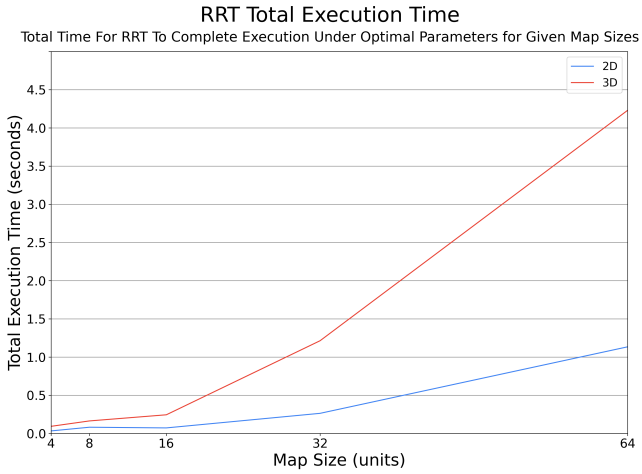


Figure B.3: **Increasing Total Execution Time of RRT with Map Size**

Appendix C

HoneyBee Supporting Documentation

C.1 Prior Work in 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.

Acceleration of Motion Planning

Accelerating motion planning with hardware is a fairly well studied problem.
A Motion Planning Processor on Reconfigurable Hardware [19] 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* [20], 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 [21] 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* [22], 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.

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.

C.2 Technical Specifications for Edge Collision Unit

Element	Requirement	Description/Justification
Performance		
Latency (μseconds/edge)	0.9	3 times faster than benchmark CPU latency.
Throughput (edges/second)	1,111,111	follows from latency.
Area		
FPGA Area	274,080 LUTs	Maximum number of LUTs on a Kintex-7 Low Voltage FPGA
Interface		
Constraints		
Length ϵ	4	ϵ defines the max edge length. The space being checked and the output sequence has the dimensions $\epsilon \times \epsilon \times \epsilon$ 4 was chosen after determining what was feasible for given area.
Inputs		
Edge e	$\{(x_1, y_1, z_1), (x_2, y_2, z_2)\}$	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	`clk, rst, start`	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	Grids are represented by their index in the sequence. 1 if collides with grid at that index, 0 otherwise.
Control Outputs	`done, idle, ready`	Output ports for control signals: idle, done, ready. These are required for adding the unit to a processor. These ouput signals are part of the "handshake" interface protocol.

Table C.1: **Full Technical Specifications for Edge Collision Detection Unit**

C.3 IEEE Standard for Floating-Point Arithmetic

Integers can be represented in binary very easily. The LSB carries the value of 2^0, the next, 2^1, and so on. Examples shown below for a 32-bit integer:

00000000000000000000000000000001 = 1
00000000000000000000000000000100 = 4
00000000000000000000000000010101 = 21

Representing a floating-point fractional number in binary is also simple. Consider the fractional number 1.75. It can be represented in point binary as 1.11. This is converted back into decimal using the same logic as above:

$$2^0 + 2^{-1} + 2^{-2} = 1.75$$

However, in computers, there is no way to represent the decimal point. All that is available is a (32-bit, for example) sequence of 1s or 0s. How does a processor know where to put the decimal point? IEEE754 is a technical standard for representing floating-point numbers and conducting floating-point arithmetic. A 32-bit floating-point number is represented by a 1-bit sign, an 8-bit exponent, and a 23-bit fraction, or "mantissa". The sign is simple; 0 for a positive number, 1 for a negative number. Now consider the fractional binary number 1111.11 (= 15.75). Similarly to regular scientific notation, this can be represented with an exponent:

$$1111.11 = 1.11111 \times 2^3$$

Since we are working with binary values, we use base 10. To prove this:

$$1.11111 \rightarrow \left(2^0 + 2^{-1} + 2^{-2} + 2^{-3} + 2^{-4} + 2^{-5}\right) \times 2^3 = 1.96875 \times 2^3 = 15.75$$

In this same way, the mantissa is multiplied by 2 to the power of the exponent and the sign is added. The mantissa has an implied leading 1, so the 23 bits shown are those to the right of the decimal point. Examples are included below:

sign	exponent	mantissa	Decimal
0	01111111	1100000 00000000 00000000	1.75
1	01111111	1100000 00000000 00000000	-1.75
0	01111111	0000000 00000000 00000000	1
0	10000001	0110000 00000000 00000000	5.5

Table C.2: **IEEE-754 Floating Point Examples**

C.4 Mapping HoneyBee's Output Sequence to a Grid-Map

For an $\epsilon \times \epsilon \times \epsilon$ grids, the grid space can be represented by an ϵ^3 bit sequence. A grid's index in the sequence is a function of its coordinates. This can be thought of as a 3 dimensional nested loop, incrementing most frequently on the x-axis, then the y-axis, then the z-axis. An index of 0 is the LSB and an index of $\epsilon^3 - 1$ is the MSB.

Mathematically, this can be thought of as the following:

$$f(x, y, z) = x + \epsilon y + \epsilon^2 z$$

In HoneyBee, this can computed efficienty by using shifts rather than multiplications, as long as ϵ is an even number. For $\epsilon = 2$, SHAMT $= 1$.

$$f(x, y, z) = x + (y << \text{SHAMT}) + (z << \text{SHAMT} << \text{SHAMT}$$

Graphically, this can be represented as shown in Figure C.1 for $\epsilon = 2$.

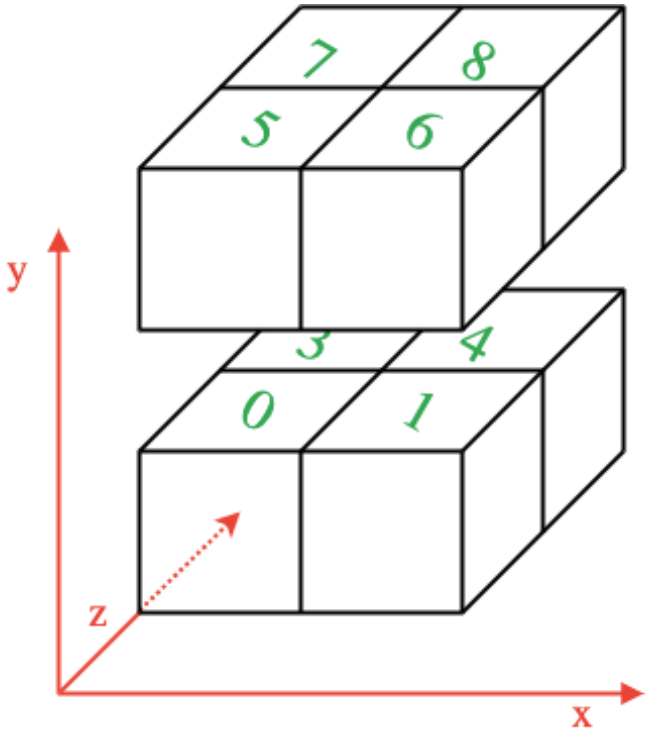


Figure C.1: **Bit Sequence Mapping for a $2 \times 2 \times 2$ Grid Space**

C.5 HoneyBee Handshake Control Protocol

Port	Description
`start`	This signal controls the block execution and must be asserted to logic 1 for the design to begin operation. It should be held at logic 1 until the associated output handshake `ready` is asserted. When `ready` goes high, the decision can be made on whether to keep `start` asserted and perform another transaction or set `start` to logic 0 and allow the design to halt at the end of the current transaction. If `start` is asserted low before `ready` is high, the design might not have read all input ports and might stall operation on the next input read.
`ready`	This output signal indicates when the design is ready for new inputs. The `ready` signal is set to logic 1 when the design is ready to accept new inputs, indicating that all input reads for this transaction have been completed. If the design has no pipelined operations, new reads are not performed until the next transaction starts. This signal is used to make a decision on when to apply new values to the inputs ports and whether to start a new transaction should using the `start` input signal. If the `start` signal is not asserted high, this signal goes low when the design completes all operations in the current transaction.
`done`	This signal indicates when the design has completed all operations in the current transaction. A logic 1 on this output indicates the design has completed all operations in this transaction. Because this is the end of the transaction, a logic 1 on this signal also indicates the data on the `return` port is valid. Not all functions have a function return argument and hence not all RTL designs have an `return` port.
`idle`	This signal indicates if the design is operating or idle (no operation). The idle state is indicated by logic 1 on this output port. This signal is asserted low once the design starts operating. This signal is asserted high when the design completes operation and no further operations are performed.

Table C.3: **Description of the ports associated with the handshake protocol for HoneyBee**. Adapted from Vivado HLS User Guide[23]

C.6 HoneyBee Interface Synthesis Report

RTL Ports	Direction	Bits	Protocol	C Type
ap_clk	in	1	ap_ctrl_hs	return value
ap_rst	in	1	ap_ctrl_hs	return value
ap_start	in	1	ap_ctrl_hs	return value
ap_done	out	1	ap_ctrl_hs	return value
ap_idle	out	1	ap_ctrl_hs	return value
ap_ready	out	1	ap_ctrl_hs	return value
ap_return	out	64	ap_ctrl_hs	return value
edge_p1_x	in	32	ap_none	scalar
edge_p1_y	in	32	ap_none	scalar
edge_p1_z	in	32	ap_none	scalar
edge_p2_x	in	32	ap_none	scalar
edge_p2_y	in	32	ap_none	scalar
edge_p2_z	in	32	ap_none	scalar

Table C.4: **HoneyBee Interface Synthesis Report** for $\epsilon = 4$. Results taken from Vivado HLS synthesis report.

C.7 HoneyBee-B Variants

HoneyBee-A

Calling each Line Intersects plane from within the honeybee() function. max latency = 6.66 us and no overlap between checking each plane. No Pragmas used, one instance of the `Check_Planes` module, computing collisions for xy, xz, and yz oriented planes sequentially.

HoneyBee-B1

By tweaking the way the C Code was laid out, 3 different instances of the same `Check_Planes` module were instantiated, executing in parallel.

HoneyBee-B2

Added HLS PRAGMA "function_instantiate" to minimise control logic depending on inputs. Instantiate an instance of 3 slightly different modules: `Check_Planes_XY`, `Check_Planes_XZ`, and `Check_Planes_YZ` Marginally faster but significantly smaller.

HoneyBee-B3, B4, B5

HB-4, HB-5, and HB-6 were all attempts at using the HLS UNROLL Pragma to unroll the ϵ deep loop for checking a set of planes, to not much avail. This failed to result in much of a speedup because the loop writes to the collision variable.

HoneyBee-C

Use HLS PRAGMA "PIPELINE" to execute the ϵ iterations in parallel.

Appendix D

Xedgcol Non-Standard Extension for Edge Collision Detection

This chapter describes version 1.0 of the Xedgcol Non-Standard ISA Extension for RISC-V.

The Xedgcol extension is designed to be implementable alongside any base ISA without reliance on any other extensions. For example, the Xedgcol instruction set requires support for floating point registers, which are not provided in the RV32I ISA. As such, it defines 6 new registers specifically for use with this ISA.

Xedgcol is a *highly* specialized ISA extension, designed explicitly for accelerating motion planning. Specifically, it is designed for the invocation of logic to compute the intersections of an edge with a grid space. The maximum length of the edge is 4 times the length of a single grid. This is relevant as a $4x4x4$ gridspace can be saved to two 32-bit x-registers.

D.1 Xedgcol Register State

The Xedgcol extension defines 6 new 32-bit floating-point registers, `e0-e5`. The term XELEN is used to desrive the width of these floating-point registers in the RISC-V ISA, and XELEN=32 for this extension. Table D.1 shows the additional register state defined by Xedgcol.

Register	ABI Name	Description
e0	px0	X coordinate of edge's first point
e1	py0	Y coordinate of edge's first point
e2	pz0	Z coordinate of edge's first point
e3	px1	X coordinate of edge's second point
e4	py1	Y coordinate of edge's second point
e5	pz1	Z coordinate of edge's second point

Table D.1: **Xedgcol Register State**

D.2 Referencing Xedgcol Registers

Since only 6 registers are defined and they are only referenced by the instructions defined in this ISA, it is possible for them to be referenced using only 3-bit values. This allows for more bits in the instructions to be used for immediate values.

D.3 Load Immediate Edge Instruction

The Load Immediate Edge (LI.e) Instruction allows for a floating point number to be loaded directly into the register *rd*. LI.e loads a single-precision value into the specified register.

imm[31:6]	rd	000
26	3	3
Floating-Point Immediate	*dest*	LI.e

Only the 26 MSB of the floating-point immediate are stored in the instruction. This should then be extended in the instruction decode stage for storage in the destination register.

D.4 Edge Collision Instruction

The Edge Collision Instruction computes the grids with which the edge defined by registers `e0-e5` collides. The result is a 64-bit sequence of collision bits that can be saved in two 32-bit destination registers.

000000000000	rd2	000	rd1	0000	100
12	5	3	5	4	3
null	*dest2*	null	*dest1*	null	ECOL

The ECOL instruction was structured the way it was in order to simplify instruction decoding in a processor. The 32 LSB are stored in rd1. The way the instruction is structured means that this can occur in the normal writeback stage of a processor with minimal extra logic. Similarly, only minimal extra instruction decoding and control logic needs to be implemented to facilitate saving the 32 MSB to rd2.

Appendix E

PhilosophyV Supporting Documentation

E.1 Reduced Instruction Set Computer (RISC)

There are two broad classifications of ISAs: Complex Instruction Set Computer (CISC) and Reduced Instruction Set Computer (RISC). Table E.1 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 E.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.

E.2 PhilosophyV Core Schematic for RV32I

See Page 89.

E.3 PhilosophyV Core Schematic for RV32I_Xedgcol

See Page

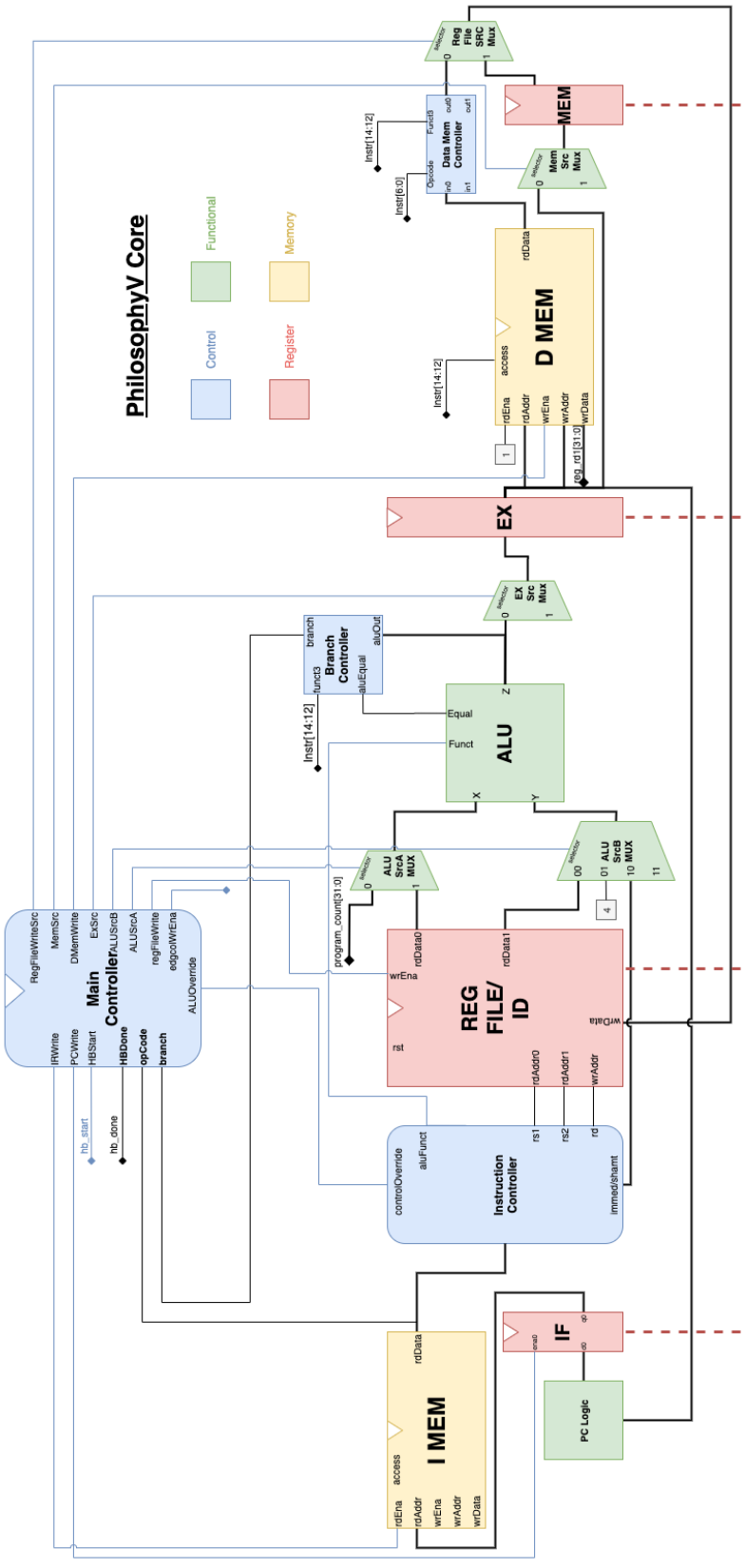


Figure E.1: **RV32I PhilosophyV Schematic**

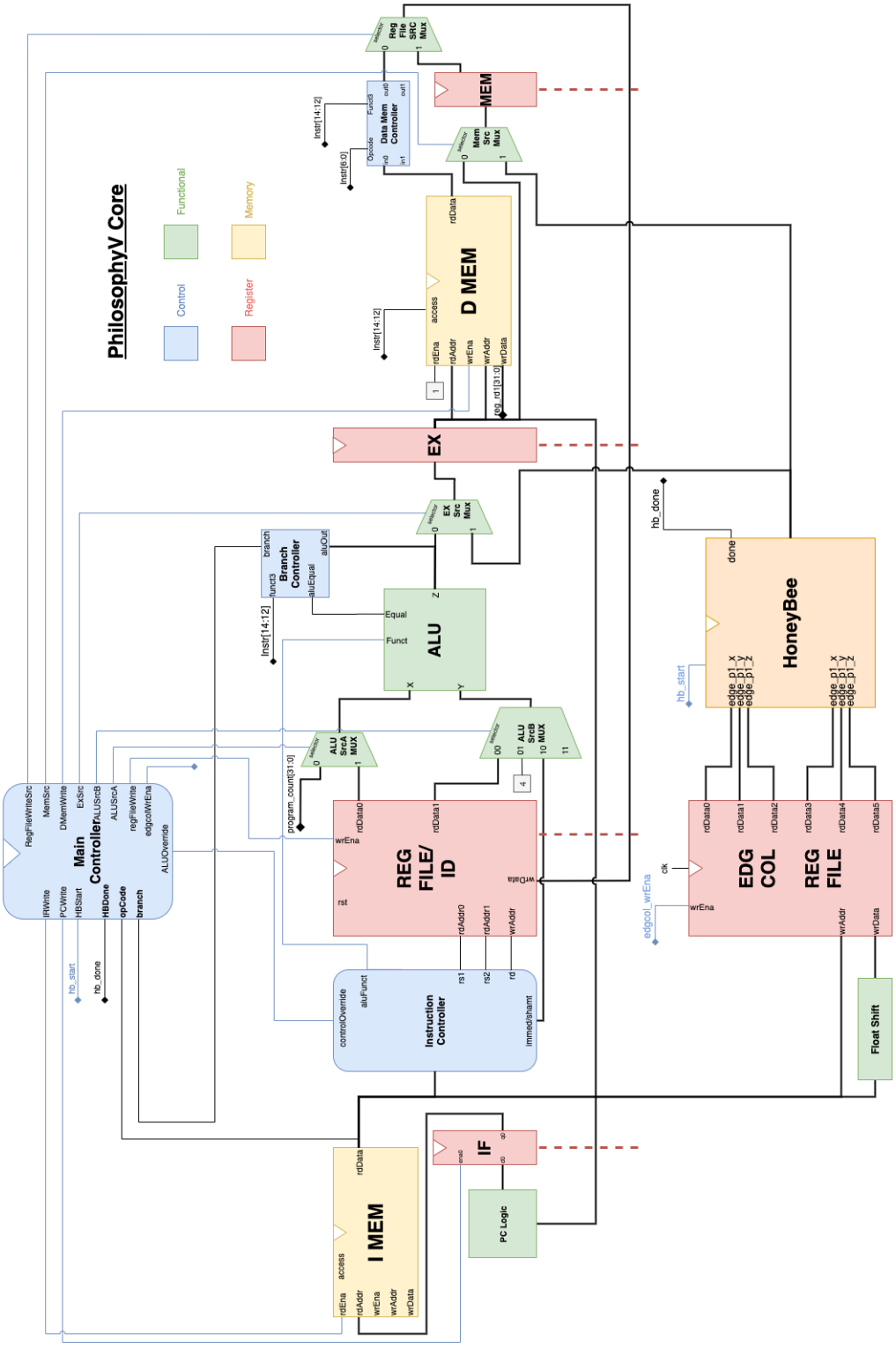


Figure E.2: **RV32I_Xedgcol PhilosophyV Schematic**

Bibliography

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

[2] 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.

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

[4] C. E. Shannon, "XXII. Programming a computer for playing chess," *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science*, vol. 41, pp. 256–275, mar 1950.

[5] M. Campbell, A. J. Hoane, and F. H. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, pp. 57–83, jan 2002.

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

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

[8] Intel, "VTune Profiler," 2019.

[9] R. Menzel, U. Greggers, A. Smith, S. Berger, R. Brandt, S. Brunke, G. Bundrock, S. Hülse, T. Plümpe, F. Schaupp, E. Schüttler, S. Stach, J. Stindt, N. Stollhoff, and S. Watzl, "Honey bees navigate according to a map-like spatial memory," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 102, pp. 3040–3045, feb 2005.

[10] "The man who knew too much: Alan Turing and the invention of the computer," *Choice Reviews Online*, 2006.

[11] F. Elsabbagh, B. Asgari, H. Kim, and S. Yalamanchili, "Vortex RISC-V GPGPU System: Extending the ISA, Synthesizing the Microarchitecture, and Modeling the Sooware Stack," tech. rep., 2019.

[12] Chips Alliance, "Rocket Chip," 2020.

[13] Thingbits, "LynxMotionHQuad500 Drone."

[14] RoboJackets, "RRT," 2019.
https://github.com/RoboJackets/rrt.

[15] M. Planning, "rrt-algorithms," 2019.
https://github.com/motion-planning/rrt-algorithms.

[16] Sourishg, "rrt-simulator," 2017.
https://github.com/sourishg/rrt-simulator.

[17] Vss2sn, "Path Planning," 2019.
https://github.com/vss2sn/path_planning.

[18] Olzhas, "RRT Toolbox," 2017.

[19] 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.

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

[21] 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.

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

[23] Xilinx, "Vivado Design Suite User Guide for High-Level Synthesis," tech. rep., 2014.

BOM (Bill of Materials)	Unit Cost $	Unit	# of Units	Total $	Exact or estimated?
No materials were purchased in the production of					Exact
this thesis					

Please do your best to fill in the following details on	Total $		Exact or estimated?
Total Development cost: everything that was spent	0		Exact
What is the minimum cost to make one prototype of	0		
Total cost of items purchased through the *Active*	0		
Total cost covered by the *Harvard Research*	0		
Total cost of items purchase personally, if any	0		
Total cost covered by a *non-Harvard lab* and/or	0		

If you had to use CNS space and equipment please answer the
Why did your project require the use of CNS facilites?
Roughly much time did you spend using CNS facilities? [specify hours or days]
What equipment did you need to use?
Who did you interact with at CNS and how?
If you are affiliated with a research lab that arranged for your use of
If you didn't work through a research lab directly, did the Active
What could we have done better to make your CNS experience more

Please list below all material used in **ALL**