

IF GOINGToCrash(): DONT()
RISC-V ARCHITECTURE FOR MOTION PLANNING
ALGORITHMS IN AUTONOMOUS UAVs

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

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

Faculty Advisor: Vijay Janapa Reddi

Harvard University School of Engineering and Applied Sciences
Cambridge, MA

March 1st, 2020
Version: 4.2

Fix Cover Page Formatting

DRAFT

Abstract

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

Rewrite Abstract

Acknowledgements

Contents

Preface	i
Abstract	i
Table of Contents	ii
List of Acronyms	v
List of Algorithms	vii
List of Figures	viii
List of Tables	x
1 Introduction	1
1.1 Problem Summary	1
1.1.1 Background	1
1.1.2 Problem Definition	4
1.2 Prior Work	4
1.2.1 Hardware Acceleration	4
1.2.2 RISC-V	6
1.3 Project Overview	8
1.3.1 Proposed Solution	8
1.3.2 Project Specifications	9
1.3.3 Project Structure	9
2 Motion Planning in Software	11
2.1 Motion Planning Background	11
2.1.1 Key Concepts	12
2.1.2 Rapidly-exploring Random Tree	14
2.2 Implementation of RRT	18
2.2.1 Technical Specifications	18
2.2.2 Implementation Design	19
2.2.3 Implementation Visualization	20
2.3 Analysis of RRT	23
2.3.1 Experimental Methodology	23

2.3.2	Results	24
3	Motion Planning in Hardware	28
3.1	Defining the Collision Detection Unit	28
3.1.1	Edge Collision Function	28
3.1.2	Technical Specifications	30
3.2	HoneyBee	32
3.2.1	HoneyBee Interface Design	33
3.2.2	HoneyBee Implementation	35
3.2.3	HoneyBee Acceleration	39
4	Motion Planning Architecture	45
4.1	Computer Architecture Background	45
4.1.1	Instruction Set Architecture	46
4.1.2	Reduced Instruction Set Computer (RISC)	46
4.1.3	RISC Microarchitecture	47
4.2	RISC-V Instruction Set	48
4.2.1	RISC-V	48
4.2.2	RV32I	50
4.2.3	Defining a RISC-V Custom Extension	51
4.3	PhilosophyV	53
4.3.1	RV32I Implementation	53
4.3.2	RV32I_Xedgcol Implementation	54
4.3.3	Verification and Analysis	56
5	Conclusion	58
5.1	Discussion of Results	58
5.2	Evaluation of Success	58
5.3	Future Work	58
	Bibliography	59
	Glossary	61
	Appendices	65
A	Project Repository	66
B	Budget	67

C	RRT Supporting Documentation	68
C.1	Justification of Modelling UAV as Prism	68
C.2	Full Technical Specifications for RRT Implementation	69
C.3	Assessment of Existing RRT Implementations	70
C.4	Implementation of Key RRT Functions	71
C.5	Geometrically Determining Segment-Plane Intersection	74
C.6	Timing Methodology of RRT Analysis	75
D	HoneyBee Supporting Documentation	77
D.1	Technical Specifications for Edge Collision Unit	78
D.2	IEEE Standard for Floating-Point Arithmetic	79
D.3	Mapping HoneyBee's Output Sequence to a Grid-Map	80
D.4	HoneyBee Handshake Control Protocol	81
D.5	HoneyBee Interface Synthesis Report	82
D.6	HoneyBee-B Variants	82
E	Xedgcol Non-Standard Extension for Edge Collision Detection	84
E.1	Xedgcol Register State	84
E.2	Referencing Xedgcol Registers	85
E.3	Load Immediate Edge Instruction	85
E.4	Edge Collision Instruction	85
F	PhilosophyV Supporting Documentation	86
F.1	PhilosophyV Design for RV32I	86
F.2	PhilosophyV Core Schematic for RV32I	86
F.3	PhilosophyV Design for RV32I_Xedgcol	86
F.4	PhilosophyV Core Schematic for RV32I_Xedgcol	86

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

CPI Cycles Per Instruction

CPU Central Processing Unit

CSV Comma Separated 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

2.1	Rapidly-Exploring Random Tree in Free Configuration Space	15
2.2	Rapidly-Exploring Random Tree with Collision Detection	17
C.1	getRandomConfig() as implemented for RRT	71
C.2	findNearestConfig() as implemented for RRT	71
C.3	stepFromNearest() as implemented for RRT	72
C.4	configCollision() as implemented for RRT	72
C.5	configCollision() as implemented for RRT for 3D	73

List of Figures

1.1	Simple Visualization of Computer Implementation Hierarchy	5
1.2	Typical Process of Adding Non-Standard Extension to RISC-V ISA	7
1.3	System Diagram of Overall Project	8
2.1	Example of 2 Robot Configurations in 3D Space for Motion Planning Purposes	13
2.2	Occupancy Grid Maps for a (16×16) Workspace of Different Resolutions . .	14
2.3	Scope of the RRT Algorithm	15
2.4	Demonstration of RRT Algorithm for 2D robot in 2D space.	16
2.5	Demonstration of the 5 Key Functions that Constitute RRT	20
2.6	Visualization of Workspace in 2D and 3D	21
2.7	Visualization of Obstacles in 2D and 3D	22
2.8	Complete Visualization of RRT in 2D and 3D	22
2.9	Increasing Total Execution Time of RRT with Map Size	25
2.10	Profile of Computational Load of RRT in 2D	26
2.11	Profile of Computational Load of RRT in 3D	27
3.1	Detecting Grid Intersections by Finding Intersections with Axis Oriented Planes	29
3.2	Edge Collision Computation Process	30
3.3	Megalong Park Honey Bee Pollinating a Weeping Cherry Blossom	32
3.4	HoneyBee in a Motion Planning Processor	33
3.5	General Overview of HoneyBee Interface	33
3.6	Port Diagram of HoneyBee Interface	34
3.7	The Impact of ϵ on the Length of the Bit-Collision Sequence	35
3.8	The Harvard Mark I Computer	36
3.9	The Hardware Development Process	36
3.10	Hardware Optimization Process	38
3.11	HB-A Performance Against Benchmark CPU	39
3.12	Timing Diagrams Showing Parallelization in HoneyBee-B	40
3.13	HB-B Performance Against Benchmark CPU	42
3.14	Timing Diagrams Showing Parallelization in HoneyBee-C	43
4.1	Overview of the Field of Computer Architecture	45
4.2	ISA as a Contract Between Software and Hardware Developers	46
4.3	5-Stage Reduced Instruction Set Computer (RISC) Datapath	47
4.4	Typical Process of Adding Non-Standard Extension to RISC-V ISA	49
4.5	RISC-V ISA Modularity	50
4.6	Simplified Schematic of the RV32I PhilosophyV Core	53
4.7	Simplified Schematic of the RV32I_Xedgcol PhilosophyV Core	54

4.8	PhilosophyV Register Files	55
4.9	Implementation of HoneyBee in RV32I_Xedgcol PhilosophyV	56
C.1	Modelling a UAV as a Rectangular Prism	68
C.2	Using Parallel Planes to determine Edge Collisions with Grids	74
D.1	Bit Sequence Mapping for a $2 \times 2 \times 2$ Grid Space	80
F.1	RV32I PhilosophyV Schematic	87
F.2	RV32I_Xedgcol PhilosophyV Schematic	88

List of Tables

1.1	List of System Components and their Descriptions	9
2.1	Abbreviated Technical Specifications for RRT Implementation	18
2.2	RRT Implementation Parameters	19
2.3	Optimal RRT Parameters for each Map Size	24
3.1	Performance Specifications for Edge Collision Detection Unit	30
3.2	Interface Specifications for Edge Collision Detection Unit	31
3.3	Synthesis Results for HB-A with $\epsilon = 4$	37
3.4	Synthesis Results for HB-B3 with $\epsilon = 4$	41
3.5	Synthesis Results for HB-C with $\epsilon = 4$	44
4.1	Comparison of CISC and RISC ISAs.	47
4.2	Required Bits to Represent Output Collisions For Different Values of Epsilon	51
C.1	General Technical Specifications for RRT Implementation	69
C.2	Required Parameters for RRT Implementation	70
C.3	Evaluation of Existing Open-Source Implementations of RRT	70
C.4	Comparison of Timing Methods	76
D.1	Full Technical Specifications for Edge Collision Detection Unit	78
D.2	IEEE-754 Floating Point Examples	79
D.3	Description of the ports associated with the handshake protocol for HoneyBee	81
D.4	HoneyBee Interface Synthesis Report	82

Chapter 1

Introduction

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

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

1.1 Problem Summary

1.1.1 Background

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

cite

Autonomous Robotics

For well over 2000 years, the concept of robotics, albeit not always with such a term, has fascinated humans. As early as the first century A.D., the Greek mathematician and engineer, Heron of Alexandria, described more than 100 different machines and automata in *Pneumatica* and *Automata* [1]. In 1898, Nikola Tesla demonstrated the first radio-controlled vessel. Since then, the world has seen widespread application of robotics in manufacturing, mining, transport, exploration, and weaponry. For the last few decades, robots have operated in controlled, largely unchanging environments (e.g. an assembly line) where their environment and movements are largely known *a priori*.

However, in recent years a new generation of autonomous robots has been developed for a wide range complex applications. These new robots are required to adapt to the changing environments in which they operate; most often, this means planning their own paths through space. As such, they must perform motion planning in real-time.

Motion Planning

While most creatures in the animal kingdom find it relatively easy to navigate their surroundings, autonomous robots must be taught explicitly how to do so by their programmers. Motion Planning refers to the problem of algorithmically determining a collision-free path between two points in an obstacle-ridden space. Chapter 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 running on regular Central Processing Units(CPUs) are too slow to execute in real-time for robots 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).

cite

Application Specific Processors

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

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

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

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

Discuss Moore's law and denard scaling

RISC-V

RISC-V (pronounced “risk-five”) is an ISA developed by the University of California, Berkeley. It is established on the principles of a RISC, a class of instruction sets that allow a processor to have fewer Cycles Per Instruction (CPI) than a Complex Instruction Set Computer (CISC) (x86, the ISA on which macOS and linux operating systems run, is an example of a CISC instruction set).

What makes RISC-V unique is its open-source nature. RISC-V was started with the philosophy of creating a practical, open-source ISA that was usable in any hardware or software without royalites. The first report describing the RISC-V Instruction Set was published in 2011 by Andrew Waterman, Yunsup Lee, David A. Patterson, and Krste Asanović [2].

1.1.2 Problem Definition

Problem Statement

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

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

End User

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

Revise End User

1.2 Prior Work

Distribute All this to other parts of the report


1.2.1 Hardware Acceleration

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

Computer Implementation Hierarchy

To briefly frame the space in which this thesis operates, consider the typical computer implementation hierarchy, demonstrated in Figure 1.1. **User level applications**, such as Google Chrome, Microsoft Word, and Apple's iTunes, sit at the top of the abstraction hierarchy. These applications are implemented in **High-Mid Level Languages**, such as C/C++, Python, Java, etc. These programming languages have their own hierarchy,

but for the purpose of this thesis, it is sufficient to understand that these programming languages are then compiled into **Assembly Language**. Assembly language closely follows the execution of instructions on the **processor**, and is defined by an **ISA**. An ISA can be thought of as the contract between software programmers and processor engineers, agreeing what instructions the processor is able to implement. This assembly code is finally loaded into the processor's instruction memory and executed.



chapters/chapter1/img/computerHierarchy.png

Figure 1.1: Simple Visualization of Computer Implementation Hierarchy

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

Acceleration of Motion Planning

Accelerating motion planning with hardware is a fairly well studied problem.

A Motion Planning Processor on Reconfigurable Hardware [3] 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 Probabilistic 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* [4], Murray et al. built on the work of the aforementioned paper, to accelerate several aspects of motion planning in an efficient manner.

FPGA based Combinatorial Architecture for Parallelizing RRT [5] 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* [6], Murray et al. describe a method for constructing robot-specific hardware for motion planning, based on the method of constructing collision detection circuits for PRM that are completely parallelised, such that edge collision computation performance is independent of the number of edges in the graph. With this method, they could compute motion plans for a 6-degree-of-freedom robot more than 3 orders of magnitude faster than previous methods.

1.2.2 RISC-V

Extending RISC-V

RISC-V is designed cleverly in a modular way, with a set of base instruction sets and a set of standard extensions. As a result, processors can be designed to only implement the instruction groups it requires, saving time, space and power on instructions that won't be used. In addition, another goal of RISC-V is to provide a basis for more specialized instruction-set extensions or more customized accelerators. This is described in the most recent *RISC-V Instruction Set Manual* [7]. This is a powerful feature, as it does not break any software compatibility, but allows for designers to easily follow the steps outlined in Figure 4.4. From a hardware acceleration point of view, this is particularly useful as it allows the designer to directly invoke whatever functional unit or accelerator they implement from assembly code.



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

Accelerating RISC-V Processors

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

1.3 Project Overview

1.3.1 Proposed Solution

Revise and Rewrite

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

System Overview

Figure 1.3 shows a high level overview of the system this thesis proposes.



Figure 1.3: System Diagram of Overall Project

The **Extended RISC-V ISA** is made up of the RISC-V 32-Bit Integer (RV32I) Base Instruction Set and a **currently unnamed** on-standard extension that this thesis will define. The **PhilosophyV Processor** is a RISC-V chip built in Hardware Description Language (HDL) for this thesis. It implements both the RV32I instruction set and the non-standard extension. The PhilosophyV Core includes, along with a baseline 5-stage processor implementation, the **HoneyBee** collision detection unit. A **C Implementation of RRT** is loaded into the instruction memory of the PhilosophyV processor. This processor, synthesized on an **FPGA**, is used as the main processor, co-processor, or accelerator

on an **Autonomous UAV**. Table 1.1 outlines the components of this system and their descriptions.

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

Update Above Table

1.3.2 Project Specifications

Project Specifications: This can just be the stuff from the start. Objective and 4 goals

1.3.3 Project Structure

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

1. A benchmark motion planning algorithm, RRT, was selected and implemented in software. Once implemented, a variety of performance analysis methods were used to profile the computational hotspots of the algorithm. It was found that edge collision detection was the critical function limiting execution time. This process is detailed in Chapter 2.
2. With edge collision detection having been identified as the critical function, the process of designing specialised hardware to execute this function began. The technical specifications, performance specifications, designs, build phases, measurement and analysis of this hardware unit is presented in Chapter 3.
3. With the aforementioned functional unit's performance verified in simulation, the next step was to implement this in a processor. First, a baseline processor was designed and built for this project to implement a base RISC-V instruction set. The performance of RRT is again profiled on this baseline processor (as up until this point, it was profiled on x86 architecture). A non-standard extension to the RISC-V ISA was then defined and support for this was implemented in the processor. Comparative performance analysis was then conducted. This process is described in detail in Chapter 4.
4. Chapter 5 is a discussion of results and future work.

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

Chapter 2

Motion Planning in Software

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

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

Update
once
I have
properly
defined
goals and
objectives

2.1 Motion Planning Background

A funny paradox in computer science is the fact that it is relatively easy to teach a computer to perform tasks that humans find very complicated, but extremely difficult to program one to execute functions that humans master during infancy. Consider, it was as early as 1949 that Claude Shannon presented his paper *Programming a Computer for Playing Chess*[8], and by 1997 the *Deep Blue* computer defeated Garry Kasparov, the reigning world champion, in a six game chess match.[9] Compare that with some of the most advanced autonomous humanoid robots to date displaying dexterity only comparable with that of a toddler. The task of finding a collision free path, performed constantly without thought by a human, is an example of this paradigm. For a robot to plan its own paths, it relies on a set of Motion Planning Algorithms.

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

2.1.1 Key Concepts

Workspace

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

Configuration

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

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



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

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

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

Occupancy Grid Map

An Occupancy Grid Map (OGM) is a method of representing the obstacles present in a workspace. Obstacles are often irregularly shaped and computing collisions with such obstacles is near impossible. Therefore, the workspace is discretized into grids, with grids that contain any part of the obstacle marked as occupied, even if only a small part of the grid is occupied. An Occupancy Grid Map (OGM) will more accurately represent a workspace with a higher resolution, shown in Figure 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[10] and J. Kuffner[11]. 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.

chapters/chapter2/img/RRT-Scope.png

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
     $q_{rand} \leftarrow \text{randomConfiguration}()$ 
     $q_{near} \leftarrow \text{findNearestConfiguration}(q_{rand}, G)$ 
     $q_{new} \leftarrow \text{stepFromNearest}(q_{near}, q_{rand}, \Delta q)$ 
     $G.\text{addVertex}(q_{new})$ 
     $G.\text{addEdge}(q_{near}, q_{new})$ 
end

```

Algorithm 2.1 can be visually represented in Figure 2.4, with the example showing a 2D robot operating in a 2D workspace.



Figure 2.4: **Demonstration of RRT Algorithm for 2D robot in 2D space.** In this example, the graph G begins with an initial configuration and a goal. In (b), the first random configuration is generated. In this case, the nearest configuration is the initial 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 space will contain obstacles. As such, collision detection must be included in the algorithm. The two types of collisions the algorithm must check for are *configuration collisions* (those where the robot would collide with an obstacle in a given configuration) and *edge collisions* (where the robot would collide when moving between two collision free configurations).

RRT with configuration and edge collision detection can be seen in Algorithm 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
         $q_{rand} \leftarrow \text{randomConfiguration}()$ ;
         $q_{near} \leftarrow \text{findNearestConfig}(q_{rand}, G)$ ;
         $q_{new} \leftarrow \text{stepFromNearest}(q_{near}, q_{rand}, \Delta q)$ ;
    end
     $e_{new} \leftarrow \text{newEdge}(q_{near}, q_{new})$ 
    if !edgeCollision( $e_{new}$ ) then
         $G.\text{addVertex}(q_{new})$ ;
         $G.\text{addEdge}(q_{near}, q_{new})$ ;
    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 C.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 in 2D. Since autonomous UAVs operate in 3D space, it was necessary 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 C.1 for justification.
Mathematically Complete Collision Detection	When RRT is implemented for educational purposes, the edge collision calculations 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: <ul style="list-style-type: none"> • 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 C.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 aforementioned specifications.

2.2.2 Implementation Design

The design and implementation of RRT, while necessary, 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 C contains a more detailed account.

System
Diagram
Here

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 C.4 shows in detail how each of these functions was implemented.

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.



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



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

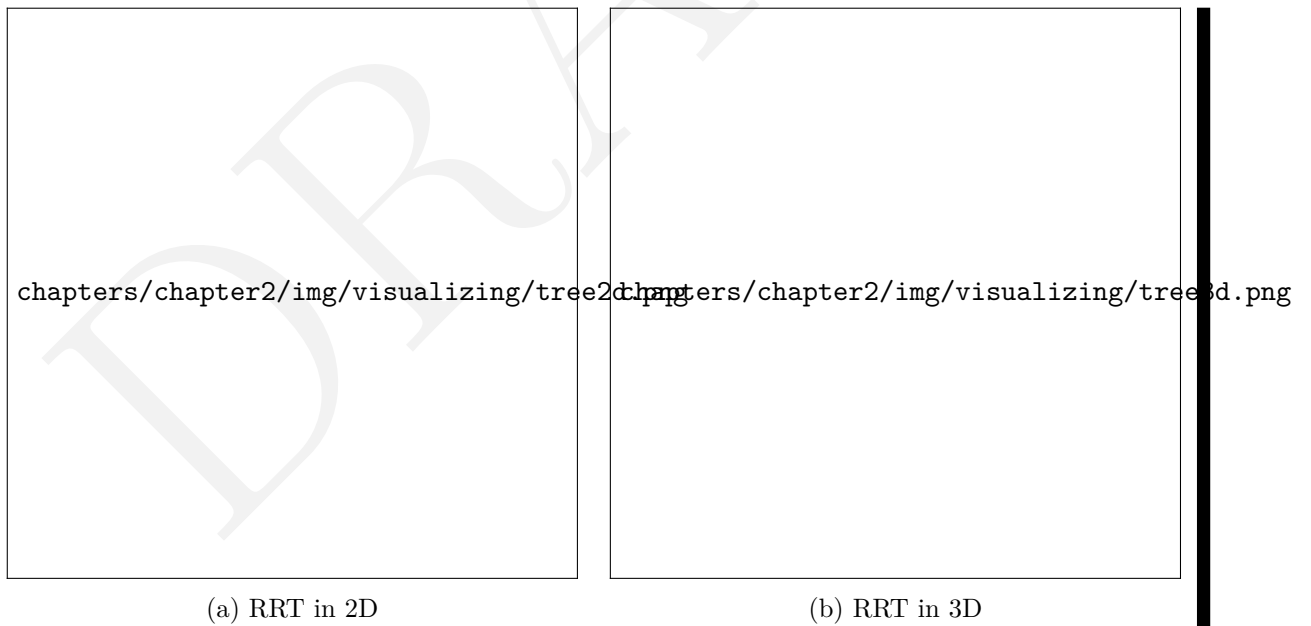


Figure 2.8: **Visualization of Obstacles in 2D and 3D**, with Graph shown in blue.

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 takes up the biggest share of computational load. The only fair way of determining the computational load of each function was to measure the percentage of CPU time each function takes for the **fastest possible execution** of RRT for a given map size. This is explained in more detail in Section 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[12]. 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 C.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, for each set of parameter values, running RRT 100 times. Thus, with 5 different map sizes, if 4 values were tested for each parameter, and 4 different OGMs were tested, the total number of tests $= 4^4 \times 5 = 1280$ (with each test running RRT 100 times!)

As such, only the optimal parameter values for each map size are 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 2.9.

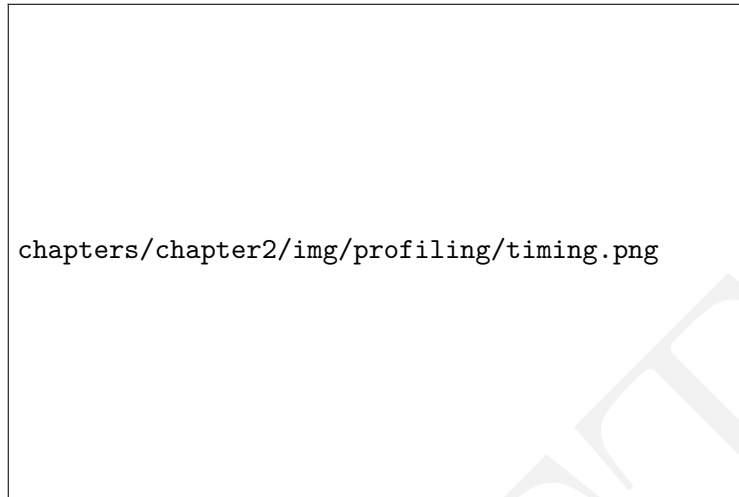


Figure 2.9: **Increasing Total Execution Time of RRT with Map Size**

2D Computational Load Profile

Figure 2.10 shows that the two biggest computational loads are `findNearestConfig()` and `edgeCollisions()`, with the latter increasing as the size of the map increases. The fact that the load of `edgeCollisions()` takes the majority of execution in bigger map sizes means that, at least in 2D, it can be considered the bottleneck function.



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

3D Computational Load Profile

The computational load of `edgeCollisions()` was even greater in 3D, starting at 40% for $4 \times 4 \times 4$ maps and increasing to 70% for $64 \times 64 \times 64$ maps, as shown in Figure 2.11.

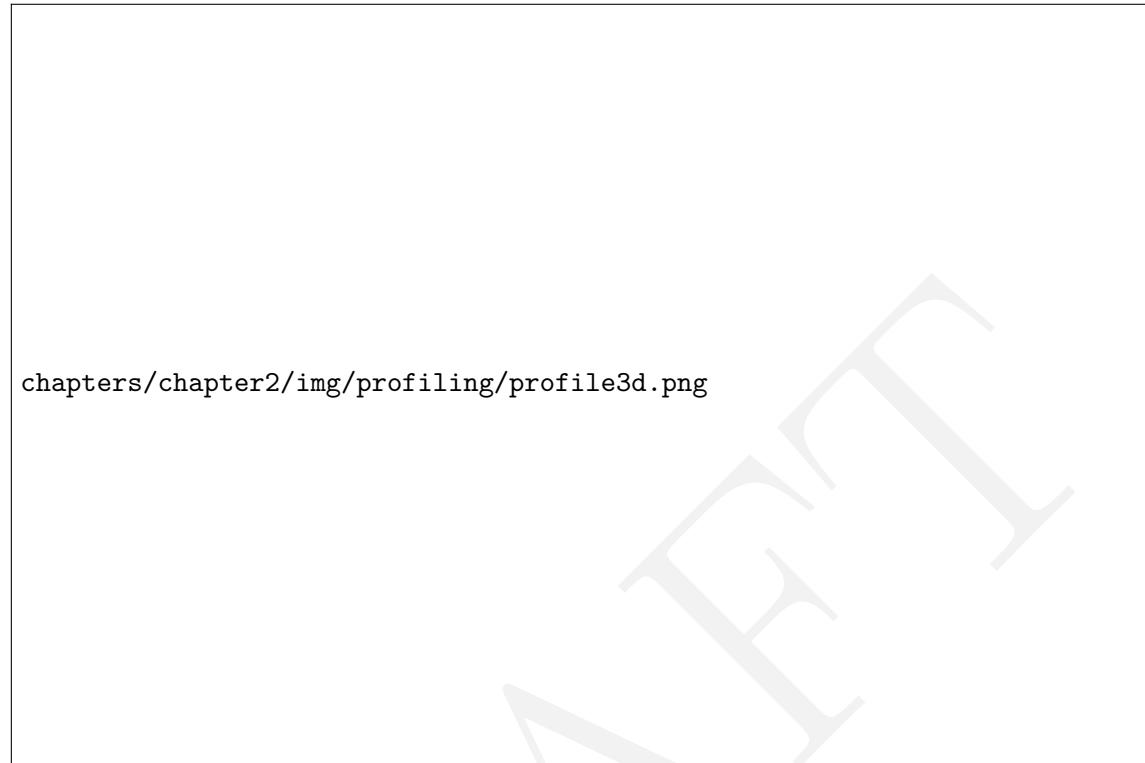


Figure 2.11: **Profile of Computational Load of RRT in 3D**

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

Chapter 3

Motion Planning in Hardware

The second objective of this thesis was to design and implement a functional hardware unit that accelerates the execution of RRT in 3D. With the bottleneck function having been identified in Chapter 2 as edge collision detection, Chapter 3 details the specification, design, implementation, and analysis of a hardware unit 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 it intersects with are “occupied”, a collision is returned. This is shown in Figure 3.2 on Page 30.

To see why it is so computationally intense to calculate intersections between a segment and grids, it must be understood that it is a fairly involved geometric process. Figure 3.1 on Page 29 shows how grid intersections are detected by computing the where the segment intersects certain **axis-oriented planes**.

Time Complexity

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

Appendix
Refer-
ence?

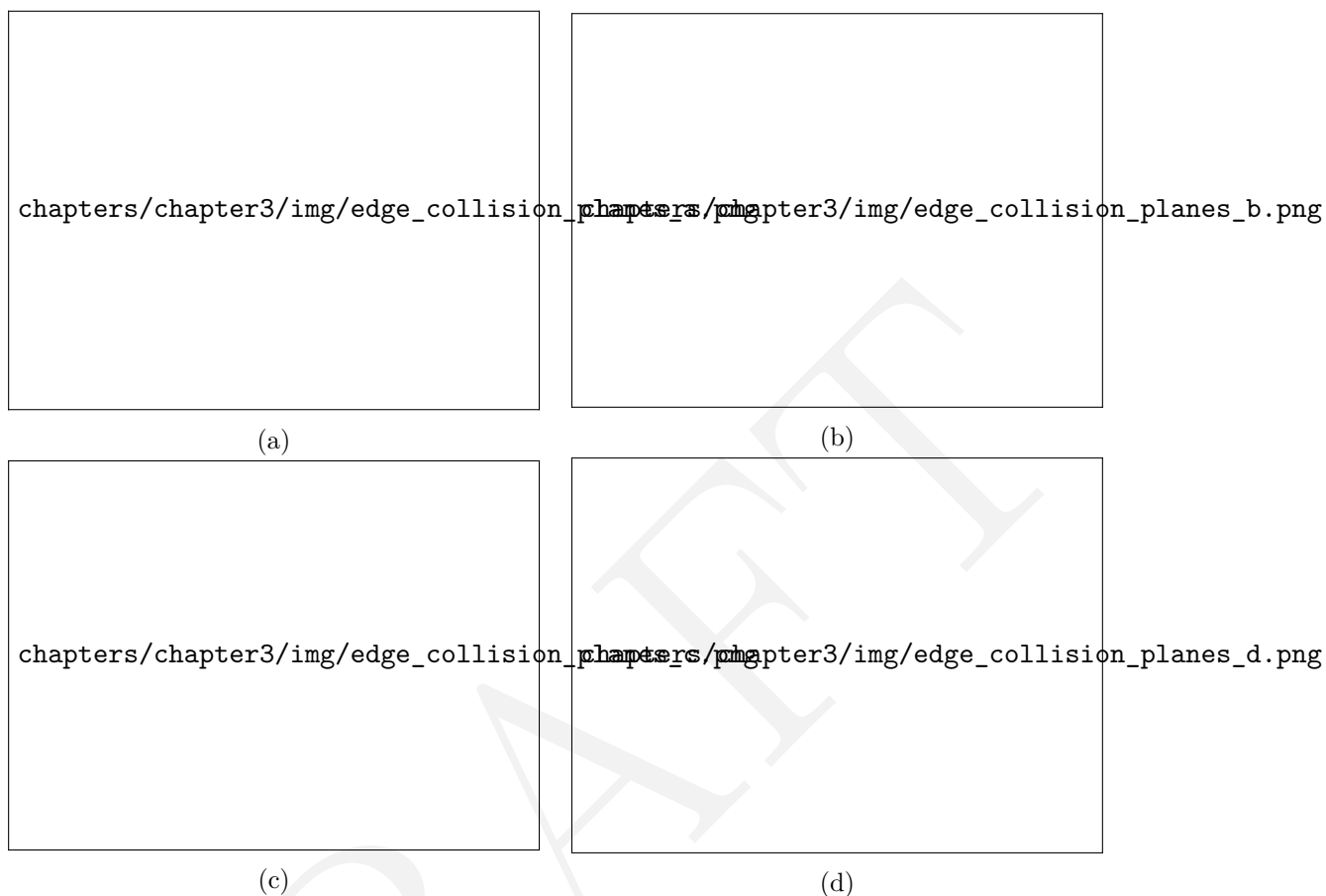
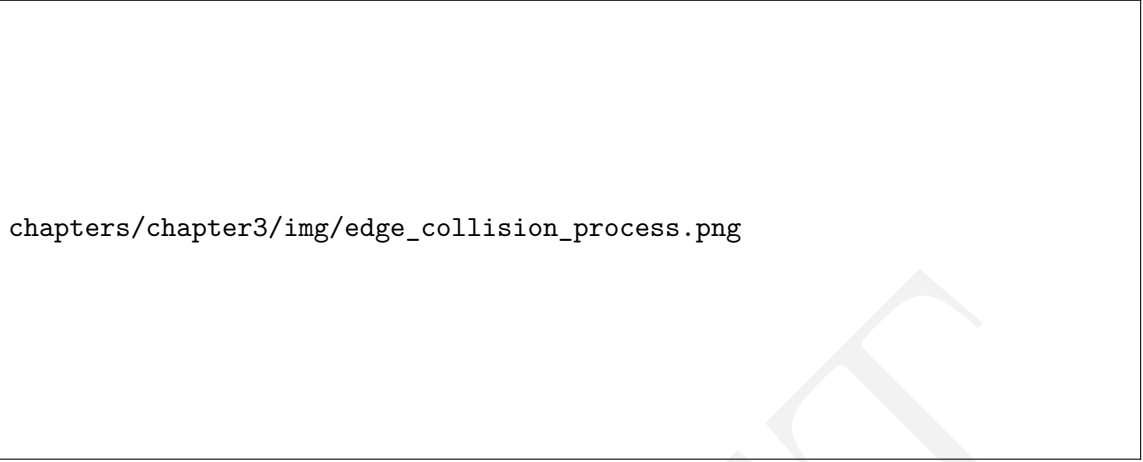


Figure 3.1: **Detecting Grid Intersections by Finding Intersections with Axis Oriented Planes**

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 executed 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.



chapters/chapter3/img/edge_collision_process.png

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

3.1.2 Technical Specifications

Performance Specifications

When accelerating motion planning algorithms, it is often difficult to quantify a goal for how much faster one would like the function to run - the answer is usually “as fast as possible!” For this thesis, the performance specification was set to be the edge collision function to run fast enough such that it was no longer the bottleneck function. This translated to a desired speedup of about 3 times (when compared to benchmark performance of a typical CPU). Table 3.1 quantifies this in terms of latency and throughput.

Metric	Benchmark CPU*	Accelerated
Latency (μ seconds/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

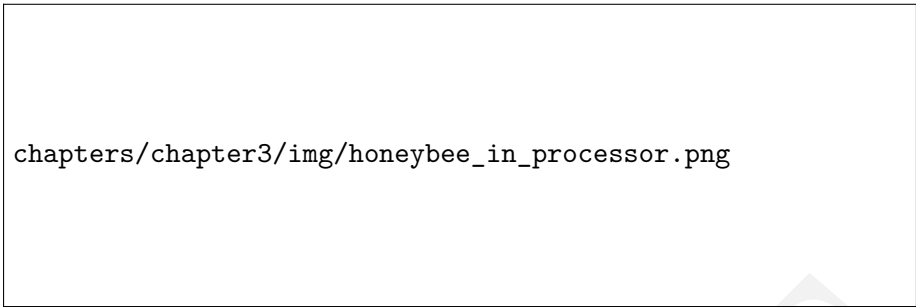
3.2 HoneyBee



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[13] 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.



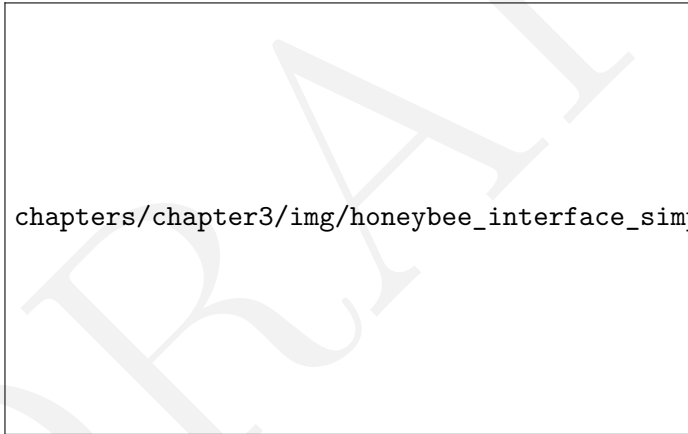
chapters/chapter3/img/honeybee_in_processor.png

Figure 3.4: HoneyBee in a Motion Planning Processor

. Shown is an abstraction of how HoneyBee would become an extension of the normal processor datapath.

3.2.1 HoneyBee Interface Design

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



chapters/chapter3/img/honeybee_interface_simple.png

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

However, when designing hardware (the method of doing so is described in section 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.

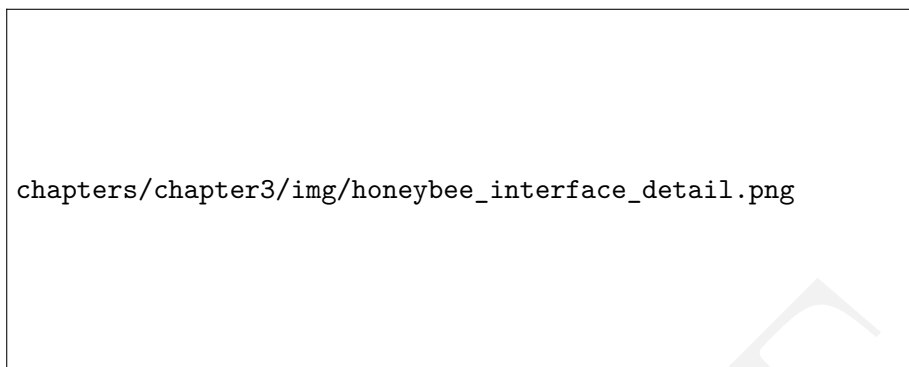


Figure 3.6: **Port Diagram of HoneyBee Interface.** The edge input is represented by 6 32-bit floats, following the IEEE 754 Single Precision 32-bit protocol, each float representing one of its coordinate points. The output sequence of collisions is a ϵ^3 -bit sequence, with each bit in the sequence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

Inputs

The inputs to HoneyBee collectively describe a single edge. This is done with 6 32-bit floating point numbers. How floating points (non-integer numbers) are represented in binary determined by the IEEE Standard for Floating-Point Arithmetic (IEEE754). How this is actually represented is not necessary to understand, but is explained in Appendix D.2. The important point is that the input edge determined by 6 32-bit coordinate points.

Output

HoneyBee outputs a sequence of “collision-bits,” with each bit in the sequence representing if the input edge collides with its corresponding bit. How this sequence of bits is mapped to a 3D grid-map is explained in Appendix D.3. 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 be. This is demonstrated in Figure ??.

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

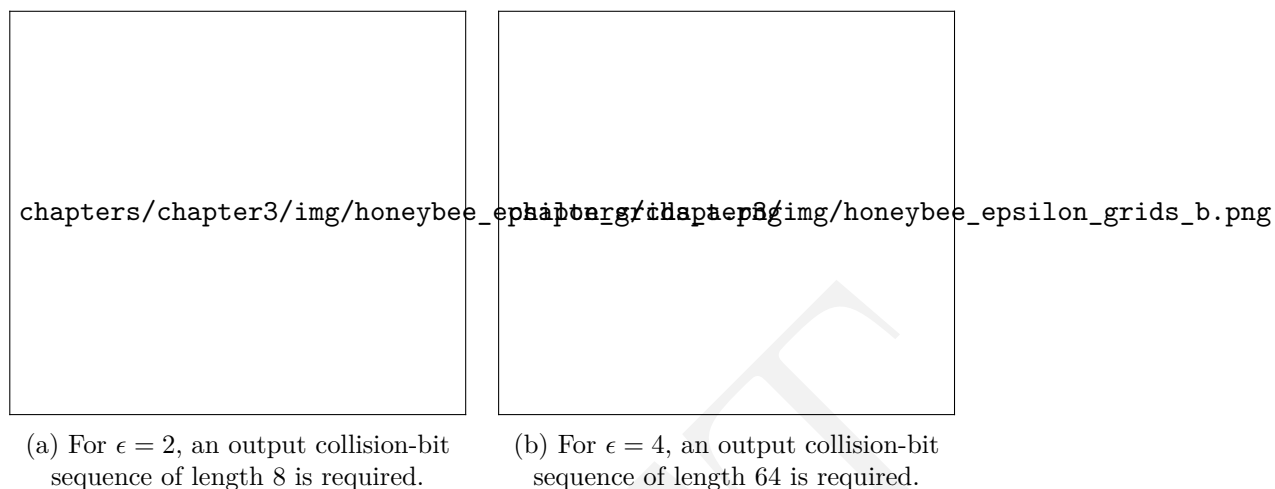


Figure 3.7: **The Impact of ϵ on the Length of the Bit-Collision Sequence**

synthesized with a certain bit-width.

Control Interface

The control interface is designed to give HoneyBee the ability to be included in a processor, and implements a commonly used “handshake” protocol between HoneyBee and the control unit of the processor in which it resides. Put simply, this is a method that allows the control unit to tell HoneyBee when to start executing the computation, and for HoneyBee to tell the control unit when it has finished its computation and the output value is ready. This is explained in detail in Appendix D.4. 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.[14] 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.[15]

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

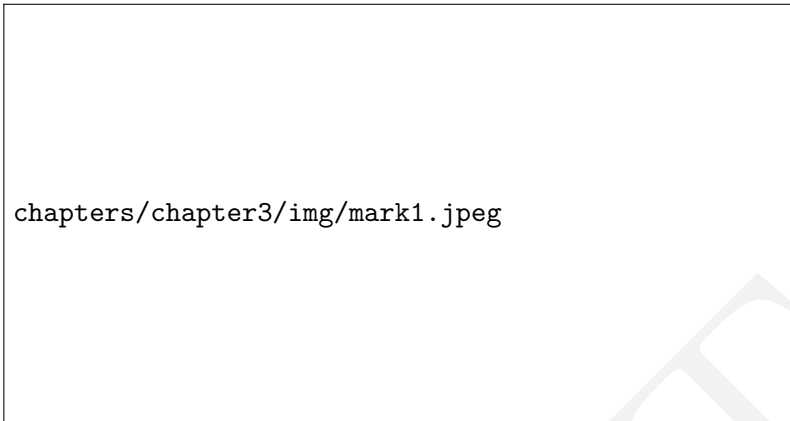


Figure 3.8: **The Harvard Mark I Computer**, photographed in the Harvard Science Center in April 2014 (Source: Harvard University)

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

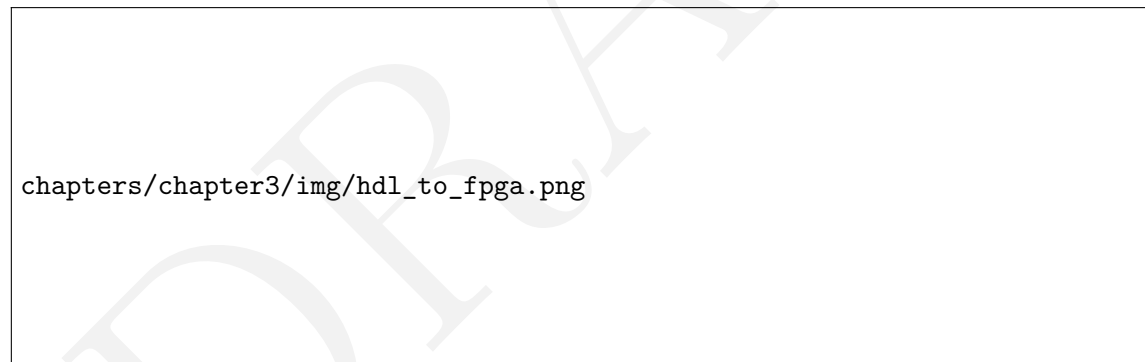


Figure 3.9: **The Hardware Development Process**, Defining Hardware Units in Hardware Description Languages for FPGAs.

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

High Level Synthesis

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

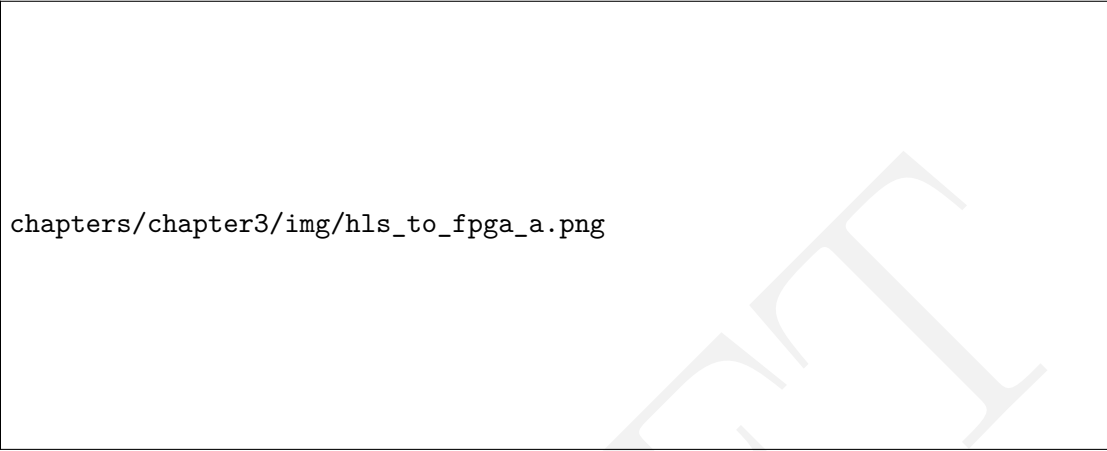
The most important benefit of using HLS, however, is the ability to use “pragmas.” These are simply directives given to the HLS tool that tell it what optimizations to use when translating C code into an HDL. This allows the same functionality to be synthesized in many different ways, optimizing the synthesis for speed, area, memory, etc. As such, the hardware development process now allows developers to experiment quickly with different ways to implement the same functionality. This is demonstrated graphically by Figure 3.10 on Page 38.

HoneyBee-A Synthesis

With the functionality of HoneyBee implemented in C, the first optimization iteration (designated HoneyBee-A (HB-A)) was synthesized. HB-A had no pragmas, and was merely a basic hardware implementation of the defined functionality. HB-A and all subsequent iterations) synthesized correctly to the interface specifications (See Appendix D.5 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 specified performance metrics. This is where the beauty of High-Level Synthesis optimization came in.

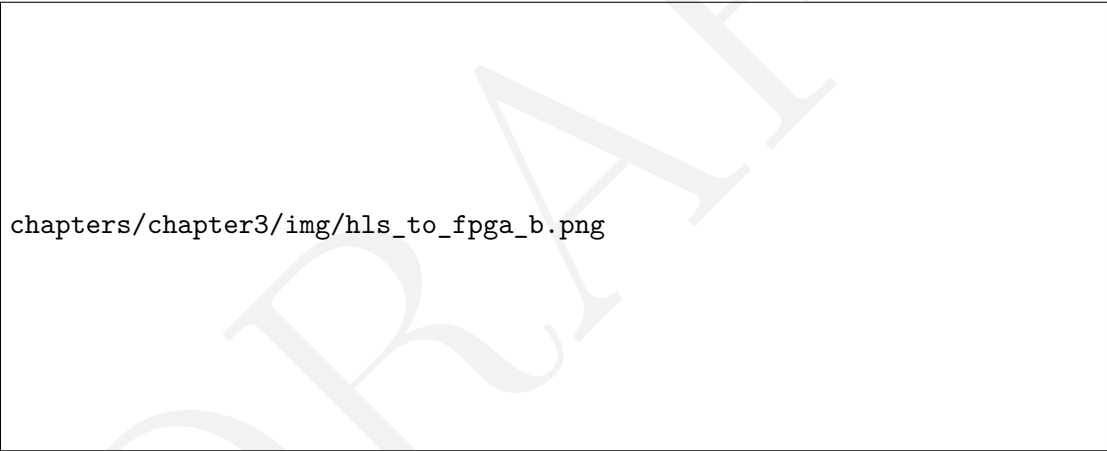
Metric	Specification	Synthesis Result
Latency (μ seconds/edge)	0.9	6.66
Throughput (edges/second)	1,111,111	150,150
FPGA Area LUTs	274,080	10,593

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



chapters/chapter3/img/hls_to_fpga_a.png

(a) Hardware Optimization without HLS



chapters/chapter3/img/hls_to_fpga_b.png

(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 achieve 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 rigorous 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 μ seconds, with a standard deviation of 0.1 μ seconds. 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 parallelism 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 3.14 shows how the process of checking each set of planes was done sequentially in HB-A, but are executed in parallel in HB-B.

chapters/chapter3/img/timing1.png

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

chapters/chapter3/img/timing2.png

(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 simplified 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 3.14a), 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 3.14b), 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 separate 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 3.14 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 D.6 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 (μ seconds/edge)	0.9	2.05
Throughput (edges/second)	1,111,111	487,805
FPGA Area LUTs	274,080	26,524

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



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

HoneyBee-C

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

chapters/chapter3/img/timing3.png

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

chapters/chapter3/img/timing4.png

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

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

Just focussing on the computation of the xy -oriented planes, Figure ?? shows how HB-B than having one instance of the module execute 4 times sequentially. HB-C implements greater parallelism by instantiating 4 module instances to execute sequentially. HB-C, as a result, exceeded the performance benchmark and was of an acceptable area, as shown in Table 3.5 and Figure 3.15

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

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

Figure 3.15:

Chapter 4


Motion Planning Architecture

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

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

4.1 Computer Architecture Background

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



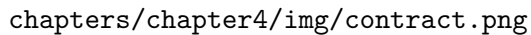
chapters/chapter4/img/architecture.png

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

4.1.1 Instruction Set Architecture

An Instruction Set Architecture (ISA) is an abstract model of a computer. On a broad level, it defines the data types, memory model, and registers of a computer, along with the instructions that it can execute.

In more human terms, it can be thought as a “contract” between hardware and software developers. It is the promise made that the hardware will be able to execute all instructions defined in the ISA, and the limitation that software must be compiled into that set of instructions.



chapters/chapter4/img/contract.png

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

The two most important things an ISA defines are a computer's **registers** and **instructions**. Instructions are the operations the computer can execute. Consider the `add` instruction: `add rd, rs1, rs2`

This instruction computes `rs1 + rs2` and stores the result in `rd`. `rd`, `rs1`, and `rs2` are all **registers**.

4.1.2 Reduced Instruction Set Computer (RISC)

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

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

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

4.1.3 RISC Microarchitecture

Figure 4.3 shows the most simple layout of a 5-stage RISC Datapath. In the **Instruction Fetch** stage, the processor gets the next instruction from memory for it to be decoded in the **Instruction Decode** stage. Here, the instruction is split into its constituent parts and has certain minor operations performed that are necessary for the next stage. The **Execution** stage is where most computation occurs. This is where the Arithmetic Logic Unit (ALU) resides, and the result of this computation goes to the **Memory** stage. This is where values are stored into or loaded from the processor's memory, these values, or the values from the Execution stage, are saved to one of the processor's registers in the **Writeback** stage.

chapters/chapter4/img/RISC-Datapath.png

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

4.2 RISC-V Instruction Set

4.2.1 RISC-V

(Pronounced “risk-five”) is an open-source and extendible ISA developed by the University of California, Berkeley. It is established on the principles of a RISC, a class of instruction sets that allow a processor to have fewer CPI than a CISC (pronounced “risk-five”) was developed at the University of California, Berkeley. It is established on the principles of RISC as an open-source and extendible 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 motivation behind designing RISC-V was largely due to the disadvantages of commercially popular ISAs.[2] (Following list adapted from RISC-V ISA Manual).

- **Commercial ISAs are proprietary.** Owners of commercial ISAs carefully guard their intellectual property and will not share implementations.
- **Commercial ISAs come and go.** Many once-popular commercial ISAs have since fallen out of fashion or are not even in production any more. Lingering intellectual property issues interfere 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.

RISC-V is designed cleverly in a modular way, with a set of base instruction sets and a set of standard extensions. As a result, processors can be designed to only implement the instruction groups it requires, saving time, space and power on instructions that won’t be used. In addition, another goal of RISC-V is to provide a basis for more specialized instruction-set extensions or more customized accelerators. This is described in the most recent *RISC-V Instruction Set Manual* [7]. This is a powerful feature, as it does not break any software compatibility, but allows for designers to easily follow the steps outlined in Figure 4.4. From a hardware acceleration point of view, this is particularly useful as it allows the designer to directly invoke whatever functional unit or accelerator they implement from assembly code.



Figure 4.4: Typical Process of Adding Non-Standard Extension to RISC-V ISA

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

Open-Source

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

Extendability

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

Modularity

Finally, RISC-V was designed to be relatively easy to implement. It is broken down into small “base” ISAs which can then have any number of standard and custom extensions added to them. These base ISAs are not designed in such a way to overarchitect for a certain type of microarchitecture. This modularity and flexibility is part of what makes it such an attractive proposition for specialized computer architecture. Figure 4.5 demonstrates this modularity.

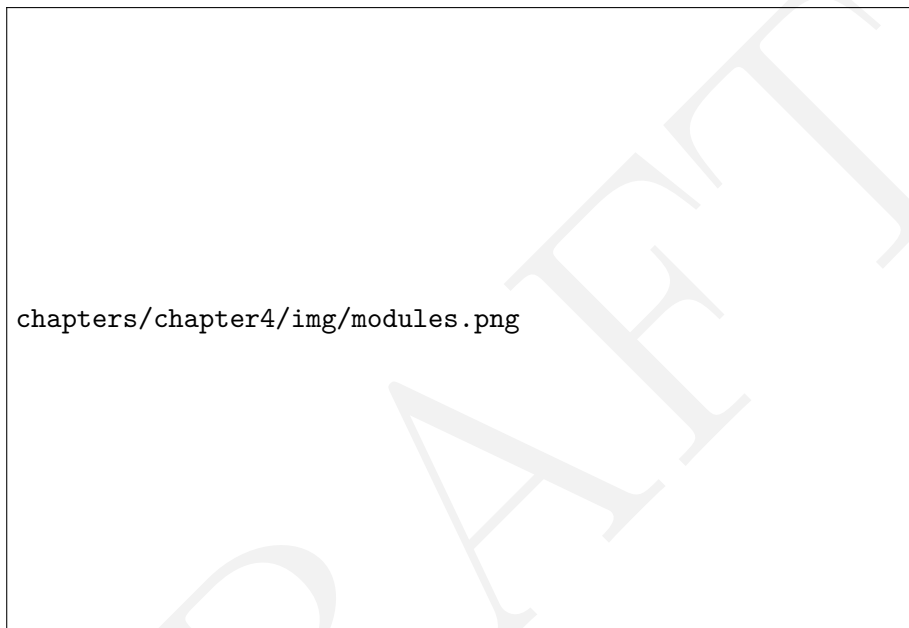


Figure 4.5: **RISC-V ISA Modularity.** It is clear that the RISC-V ISA is in fact a family of ISAs. Each is based on one of the “base” ISAs, which are each the smallest possible ISA to run just about any program. For more instructions and thus, better performance, any number of Standard or Non-Standard 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 [7]

RV32I was designed to be sufficient to form a compiler target and to support modern operating system environments. The ISA was also designed to reduce the hardware required in a minimal implementation. RV32I contains 40 unique instructions, though a simple implementation might ...[reduce] base instruction count to 38 total. RV32I can emulate almost any other ISA extension ...

Subsets of the base integer ISA might be useful for pedagogical purposes, but the base has been defined such that there should be little incentive to subset a real hardware implementation ...

4.2.3 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.

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 as $4^3 = 64$. 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. From this, it is fairly obvious why 4 was chosen 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.

I might need to make this its own section? At any rate, I need more back-ground on what an instruction looks like etc

Edge Collision Instruction

The following instruction was defined:

`ecol rd1, rd2`

It takes two destination registers and stores result of the edge collision computation in them. The 32 Least Significant Bit (LSB) are stored in `rd1` and the 32 Most Significant Bit (MSB) in `rd2`. The instruction is formatted in the following way (For more detail, the full Xedgcol specification can be found in Appendix E).

0000000000000	rd2	000	rd1	0000	001
12	5	3	5	4	3
null	<i>dest2</i>	null	<i>dest1</i>	null	ECOL

Conspicuously, the instruction specifies two destination registers, but no source registers. In other words, how does the processor know the coordinates of the edge? Also conspicuous is the fact that much of the instruction is null (meaningless zeros). Why not use this space to specify the coordinates of the edge?

Consider, an edge must be represented by 6 32-bit numbers. This information, 192 bits in total, can not all fit in a single 32-bit instruction as 6 immediate values (obviously). Could those 32-bit values be stored in existing registers and the address of these registers referenced in the instruction? Since there are 32 general purpose registers in RISC-V, their addresses are 5-bits long each. 6 5-bit addresses is 30 bits in total, which, by itself could fit in an instruction. While this couldn't be combined with the edge collision instruction, perhaps a second instruction could be defined that contained the addresses of all 6, or even 3, of the registers containing the edge's coordinates. This was decided against for a few reasons, but mainly because **storing floating-point numbers in integer registers**, while theoretically possible, is extremely bad practice. To store floating point values in registers, a new register set would need to be defined.

Edge Registers

4.3 PhilosophyV

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

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[16], a large open-source RISC-V implementation. However, the project was so sophisticated that learning its infrastructure and the necessary tool-chains became a massive project in itself. This was the case for many open-source projects. Those that lacked such sophisticated 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.

4.3.1 RV32I Implementation



chapters/chapter4/img/philv_rv32i.png

Figure 4.6: Simplified Schematic of the RV32I PhilosophyV Core

The first step was implementing a simple (a relative term) processor that implemented the RV32I Base ISA. Figure 4.6 shows a highly simplified schematic of the RV32I PhilosophyV Core. Appendix F.2 provides a detailed schematic.

The design was that of a simple 5-stage non-pipelined processor with no branch-prediction or other optimizations. In other words, it's awful. Running the original RRT program on this processor would take an inordinate amount of time (luckily, this was never tested). However, this is irrelevant. The improved performance of implementing the edge collision detection function in hardware has already been verified against an Intel processor that was designed by some of the leading minds in computer architecture. To restate, the purpose of implementing this processor was simply to prove the validity of the Xedgcol ISA extension.

The processor 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 and comparing the states of its memory and register file at the end of the programs execution to their expected states.

4.3.2 RV32I_Xedgcol Implementation



Figure 4.7: **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 ??.

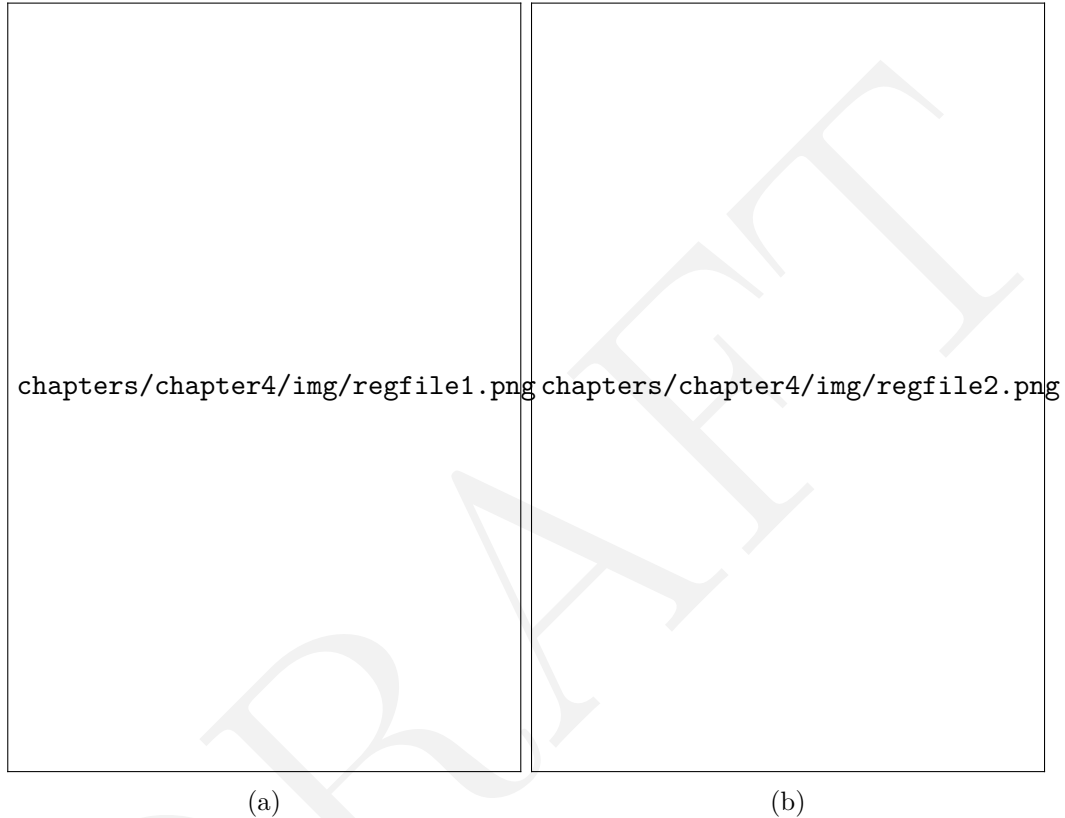



Figure 4.8: **PhilosophyV Register Files**. Figure 4.8a shows the 32RVI register file, containing 32 registers. The ports **readData0** and **readData1** output the data held at the address provided to **rdAddr0** and **rdAddr1** respectively. Figure 4.8a shows the Xedgcol register file, which is able 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.



chapters/chapter4/img/honeybee_impl.png

Figure 4.9: Implementation of HoneyBee in RV32I_Xedgcol PhilosophyV

4.3.3 Verification and Analysis

With the PhilosophyV 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.

DRAFT

Chapter 5

Conclusion

5.1 Discussion of Results

Discussion of Results

5.2 Evaluation of Success

Evaluation of Success

5.3 Future Work

Future Work

Bibliography

- [1] H. (Alexandrinus), *De gli automati, overo machine se moventi, Volume 2*.
- [2] V. I. B. U.-I. Isa, A. Waterman, Y. Lee, D. Patterson, K. Asanovi, and B. U.-I. 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] 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.
- [4] S. Murray, W. Floyd-Jones, G. Konidaris, and D. J. Sorin, "A Programmable Architecture for Robot Motion Planning Acceleration," tech. rep.
- [5] 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.
- [6] S. Murray, W. Floyd-Jones, Y. Qi, D. Sorin, G. Konidaris, and D. Robotics, "Robot Motion Planning on a Chip," tech. rep.
- [7] A. Waterman, K. Asanovic, and SiFive Inc, "The RISC-V Instruction Set Manual," vol. Volume I:, 2019.
- [8] 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.
- [9] M. Campbell, A. J. Hoane, and F. H. Hsu, "Deep Blue," *Artificial Intelligence*, vol. 134, pp. 57–83, jan 2002.
- [10] S. M. LaValle, "Rapidly-Exploring Random Trees: A New Tool for Path Planning," *In*, vol. 129, pp. 98–11, 1998.

- [11] S. M. LaValle and J. J. Kuffner, “Randomized kinodynamic planning,” *International Journal of Robotics Research*, vol. 20, pp. 378–400, may 2001.
- [12] Intel, “VTune Profiler,” 2019.
- [13] 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.
- [14] “The man who knew too much: Alan Turing and the invention of the computer,” *Choice Reviews Online*, 2006.
- [15] 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.
- [16] Chips Alliance, “Rocket Chip,” 2020.
- [17] Thingbits, “LynxMotionHQuad500 Drone.”
- [18] RoboJackets, “RRT,” 2019.
<https://github.com/RoboJackets/rrt>.
- [19] M. Planning, “rrt-algorithms,” 2019.
<https://github.com/motion-planning/rrt-algorithms>.
- [20] Sourishg, “rrt-simulator,” 2017.
<https://github.com/sourishg/rrt-simulator>.
- [21] Vss2sn, “Path Planning,” 2019.
https://github.com/vss2sn/path_planning.
- [22] Olzhas, “RRT Toolbox,” 2017.
- [23] Xilinx, “Vivado Design Suite User Guide for High-Level Synthesis,” tech. rep., 2014.

Glossary

RISC-V: (Pronounced “risk-five”) is an open-source and extendible ISA developed by the University of California, Berkeley. It is established on the principles of a RISC, a class of instruction sets that allow a processor to have fewer CPI than a CISC.

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

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-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).

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..

Configuration: A specification of a robots 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. It is obvious then that as the physical complexity of a robot increases, so too does the complexity of representing it algorithmically..

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

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

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

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

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

Real-time: Describes a system in which input data is processed in such a time period such that it is available almost immediately. Systems such as missile guidance or collision avoidance in cars are an example of real-time systems.

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

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

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

Advanced RISC Machine (ARM): A family of Reduced Instruction Set Computing architectures for computer processors, configured for various environments..

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

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..

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..

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 HDL.

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 an 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 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..

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

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

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

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..

Real-Time Operating Systems (RTOS): A type of operating system designed to operate on inputs as they come in, without buffer delays..

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..

System on Chip (SoC): A chip that includes all required components of a working computer, including one or more CPUs, memory, I/O ports, etc..

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..

DRAFT

Appendices

Appendix A

Project Repository

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

Research

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

Writeups

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

RRT

github.com/AnthonyKenny98/RRT

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

HoneyBee

github.com/AnthonyKenny98/HoneyBee

This subrepository holds the HoneyBee functional unit, a hardware implementation of collision detection.

PhilosophyV

github.com/AnthonyKenny98/PhilosophyV

This subrepository holds the PhilosophyV RISCv chip

Appendix B

Budget

Budget

Appendix C

RRT Supporting Documentation

C.1 Justification of 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.

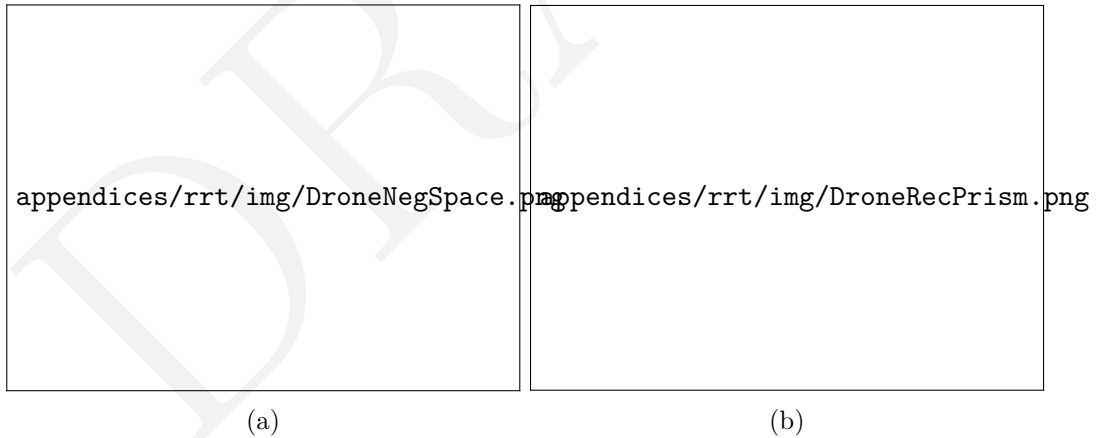


Figure C.1: **Modelling a UAV as a Rectangular Prism.** Red highlights demonstrate the configuration model, overlayed over the exact schematic. Figure C.1a shows how a drone can be modelled in high detail, but gains little useful free space when compared with Figure C.1b, which models a drone as a rectangular prism.[17]

C.2 Full Technical Specifications for RRT Implementation

General Specifications	
Requirement	Description and Justification
Implemented in C/C++	As outlined in Section 1.3.3, the critical step in determining the design of specialized hardware to accelerate RRT is CPU performance analysis of the algorithm to determine computational hot-spots. Implementations in C allow for the use of certain CPU profiling tools, 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 necessary 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 C.1 for justification of this.
Mathematically Complete Collision Detection	When RRT is implemented for educational purposes, the edge collision calculations 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 C.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 C.2: Required Parameters for RRT Implementation

C.3 Assessment of Existing RRT Implementations

Repository	General Requirements				Parameters			
	Language	2D/3D	Object Model	Collision Detect	ϵ	K	DIM	Goal Bias
RoboJackets[18]	C++	2D	Point	Complete	Yes	Yes	Yes	No
Motion-Planning[19]	Python	ND	Point	Incomplete	Yes	Yes	No	Yes
Sourishg[20]	C++	2D	Point	Incomplete	Yes	Yes	No	No
Vss2sn[21]	C++	2D	Point	Complete	Yes	Yes	No	No
olzhas[22]	Matlab	2D	Point	Complete	Yes	Yes	No	No

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

C.4 Implementation of Key RRT Functions

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

Inputs: Dimensionality N ,
 Upper Axis Bound DIM

Output: Random Configuration q

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

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

Algorithm C.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 $G.\text{existing_nodes}$ **do**
 if $\text{distance}(q_{new}, G.q[k]) < \text{distance}(q_{new}, q_{nearest})$ **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 C.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
  | |  $q_{new} \leftarrow \text{stepTowardConfig}(q_{nearest}, G.q_{goal})$ 
  | end
  | else
  | |  $q_{new} \leftarrow \text{stepTowardConfig}(q_{nearest}, q_{new})$ 
  | end
end
return  $q_{new}$ ;
  
```

Where $\text{stepTowardConfig}(q_1, q_2)$ returns a configuration ϵ from q_1 in the direction of q_2 .

Algorithm C.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  $O[\text{gridLookup}(q.x)][\text{gridLookup}(q.y)]$ 
end
else
  | return  $O[\text{gridLookup}(q.x)][\text{gridLookup}(q.y)][\text{gridLookup}(q.z)]$ 
end
  
```

Where O is a N -Dimensional array of booleans, with True representing an occupied grid and false representing an unoccupied one. $\text{gridLookup}()$ 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

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

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

Output: Boolean

```

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

```

edge e . It then looks up the OGM O to see if the grid corresponding with the point of intersection is occupied. If so, then it reports a collision by returning false. The function `edgeIntersectsPlane` follows the geometrical process of detecting a segment-plane intersection outlined in Appendix C.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 possibly 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, instead checking segment-line intersections for x and y lines.

C.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 C.2 demonstrates this point.



Figure C.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 $\langle a, b, c \rangle$ 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:

$$a(x_1 + t(x_2 - x_1)) + b(y_1 + t(y_2 - y_1)) + c(z_1 + t(z_2 - z_1)) = d$$

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.

C.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 interfere, 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 C.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 an 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 than 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 C.4: Comparison of Timing Methods

Appendix D

HoneyBee Supporting Documentation

DRAFT

D.1 Technical Specifications for Edge Collision Unit

Element	Requirement	Description/Justification
Performance		
Latency (μ seconds/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 output signals are part of the “handshake” interface protocol.

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

D.2 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:

$$\begin{aligned} 00000000000000000000000000000001 &= 1 \\ 000000000000000000000000000000100 &= 4 \\ 0000000000000000000000000000010101 &= 21 \end{aligned}$$

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:

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

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 D.2: IEEE-754 Floating Point Examples

D.3 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 be computed efficiently by using shifts rather than multiplications, as long as ϵ is an even number. For $\epsilon = 2$, SHAMT = 1.

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

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

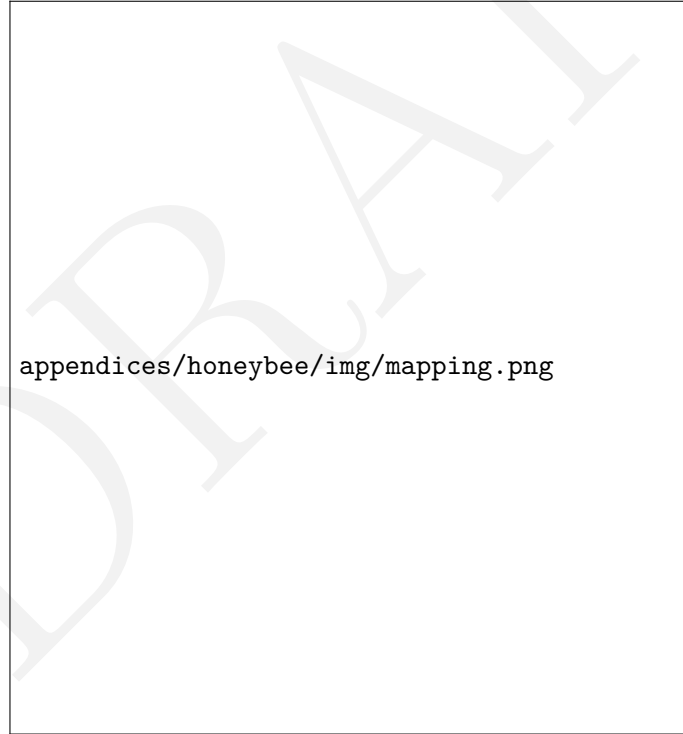


Figure D.1: Bit Sequence Mapping for a $2 \times 2 \times 2$ Grid Space

D.4 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 D.3: Description of the ports associated with the handshake protocol for HoneyBee. Adapted from Vivado HLS User Guide[23]

D.5 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 D.4: **HoneyBee Interface Synthesis Report** for $\epsilon = 4$. Results taken from Vivado HLS synthesis report.

D.6 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 E

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.

E.1 Xedgcol Register State

The Xedgcol extension defines 6 new 32-bit floating-point registers, **e0–e5**. The term XELEN is used to describe the width of these floating-point registers in the RISC-V ISA, and XELEN=32 for this extension. Table ?? 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

E.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.

E.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	<i>dest</i>	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.

E.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	001
12	5	3	5	4	3
null	<i>dest2</i>	null	<i>dest1</i>	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 F

PhilosophyV Supporting Documentation

F.1 PhilosophyV Design for RV32I

Todo

F.2 PhilosophyV Core Schematic for RV32I

See Page 88.

F.3 PhilosophyV Design for RV32I_Xedgcol

Todo

F.4 PhilosophyV Core Schematic for RV32I_Xedgcol

See Page

chapters/chapter4/img/philv-core.png

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

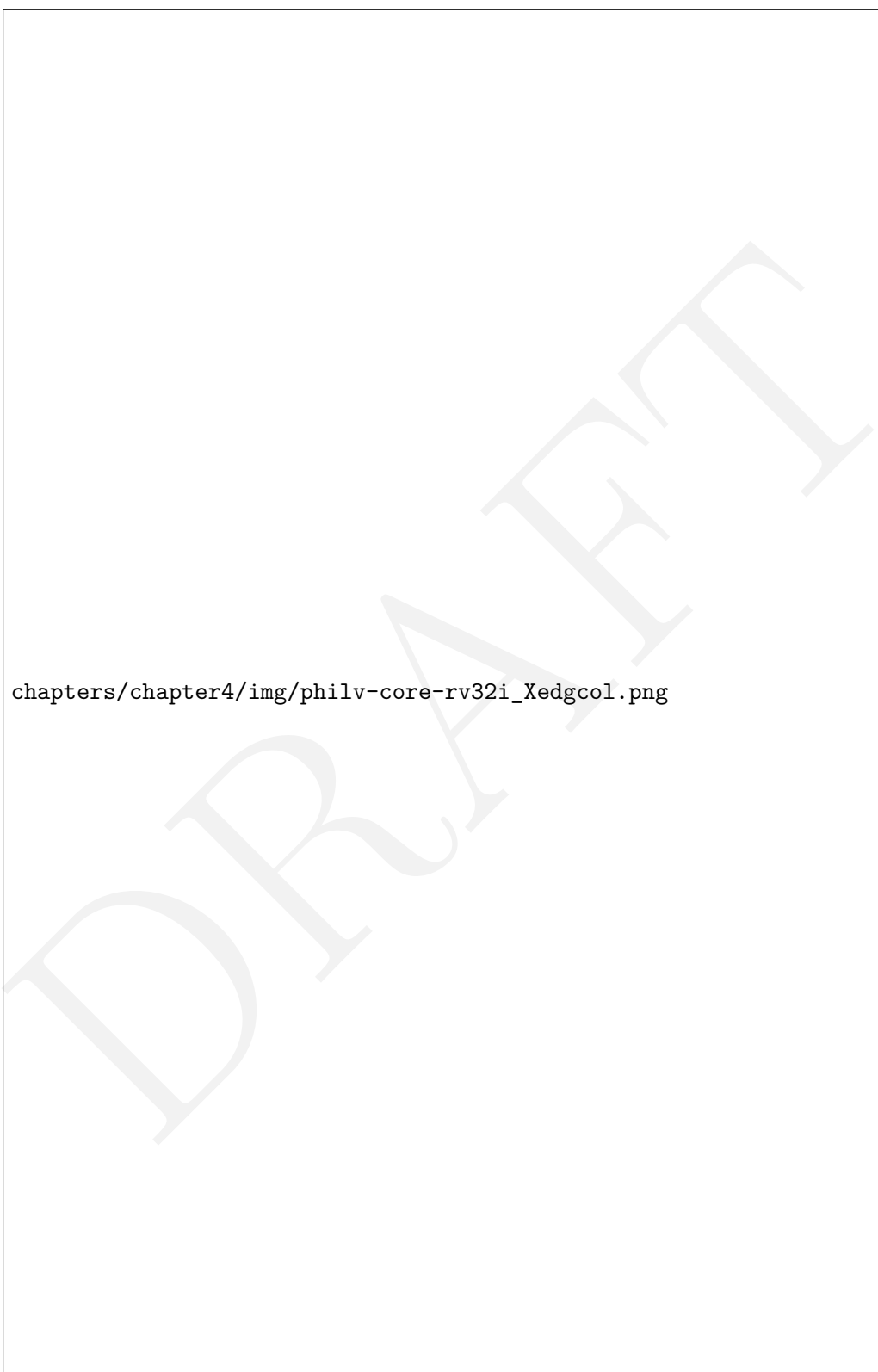


Figure F.2: **RV32I_Xedgcol PhilosophyV Schematic**

Make all reference to "Drone" uniform to "UAV" (this will require changing the "a" to "an" before each occurrence - wait actually, do I?.) Also maybe I should clarify this to rotary wing UAVs.

Spell check entire doc

Make all figure headings bold with subtitles.

DRAFT

<input type="checkbox"/>	Fix Cover Page Formatting	i
<input type="checkbox"/>	Rewrite Abstract	i
<input type="checkbox"/>	Acknowledgements	i
<input type="checkbox"/>	Define goal	1
<input type="checkbox"/>	cite	1
<input type="checkbox"/>	cite	2
<input type="checkbox"/>	Discuss Moore's law and denard scaling	3
<input type="checkbox"/>	Improve Problem Statement	4
<input type="checkbox"/>	Revise End User	4
<input type="checkbox"/>	Distribute All this to other parts of the report	4
<input type="checkbox"/>	Revise and Rewrite	8
<input type="checkbox"/>	Update Above Table	9
<input type="checkbox"/>	Project Specifications	9
<input type="checkbox"/>	Summary of Results	10
<input type="checkbox"/>	Update once I have properly defined goals and objectives	11
<input type="checkbox"/>	System Diagram Here	19
<input type="checkbox"/>	Appendix Reference?	28
<input type="checkbox"/>	I might need to make this its own section? At any rate, I need more background on what an instruction looks like etc	51
<input type="checkbox"/>	Discussion of Results	58
<input type="checkbox"/>	Evaluation of Success	58
<input type="checkbox"/>	Future Work	58
<input type="checkbox"/>	Budget	67
<input type="checkbox"/>	Todo	86
<input type="checkbox"/>	Todo	86
<input type="checkbox"/>	Make all reference to "Drone" uniform to "UAV" (this will require chaniging the "a" to "an" before each occurence - wait actually, do I?.) Also maybe I should clarify this to rotary wing UAVs.	89
<input type="checkbox"/>	Spell check entire doc	89
<input type="checkbox"/>	Make all figure headings bold with subtitles.	89