**A project report submitted in partial fulfilment for the degree of**

**BSc (Hons) Software Engineering**

**School of Psychology and Computer Science**

**University of Central Lancashire**

Programming Language for homemade CPU architecture - Gaungau

Rhys Garman

**20th April 2022**

# Abstract

Remove any text highlighted in yellow before submission.

You are aiming for no more than 50 pages of report content, this count starts at Chapter 1 and does not includes your references or appendices.

The abstract is the summary of the project report within one page (aim for about 500 words). Unnumbered chapter headings, as above, are entered using the ‘Heading (Unnumbered)’ style, which automatically starts a new page.

This template starts the page numbering at the foot of this page. That is, the first page does not have a number.

It is suggested that the abstract be structured as follows:

* Problem: What you tackled, and why this needed a solution
* Objectives: What you set out to achieve, and how this addressed the problem
* Methodology: How you went about solving the problem
* Achievements: What you managed to achieve, and how far it meets your objectives.

# Attestation

I understand the nature of plagiarism, and I am aware of the University’s policy on this.

I certify that this document reports original work by me during my University project. I also confirm that I adhere to the University’s legal and ethical guidelines for undergraduate projects in Computing.

Signature: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Date:

# Acknowledgements

Acknowledge anyone who has helped you in your work such as your supervisor, technical support staff, fellow students, or external organisations. Acknowledge the source of any work that is not your own.

# Table of Contents

[Abstract i](#_Toc107845384)

[Attestation ii](#_Toc107845385)

[Acknowledgements iii](#_Toc107845386)

[Table of Contents iv](#_Toc107845387)

[List of Figures vii](#_Toc107845388)

[List of Tables viii](#_Toc107845389)

[List of Listings ix](#_Toc107845390)

[1 Introduction 1](#_Toc107845391)

[2 Background and Related Work 2](#_Toc107845392)

[2.1 Introduction 2](#_Toc107845393)

[2.2 Teaching Low Level Programming 2](#_Toc107845394)

[2.2.1 Why Teach Programming 2](#_Toc107845395)

[2.2.2 Best Ways to Teach 2](#_Toc107845396)

[2.2.3 A practical approach 5](#_Toc107845397)

[2.3 Low-Level Programming 6](#_Toc107845398)

[2.3.1 Low-Level vs. High-Level 6](#_Toc107845399)

[2.3.2 Digital Logic 7](#_Toc107845400)

[2.3.3 Relevance of Low-Level Programming 7](#_Toc107845401)

[2.3.4 Applications of Low-Level Programming 7](#_Toc107845402)

[2.4 Microcontrollers 9](#_Toc107845403)

[2.4.1 Understanding 9](#_Toc107845404)

[2.4.2 Applications 9](#_Toc107845405)

[2.4.3 Relevance 10](#_Toc107845406)

[2.4.4 Uses in Education 11](#_Toc107845407)

[2.5 Summary 11](#_Toc107845408)

[3 Project Planning 13](#_Toc107845409)

[3.1 Introduction 13](#_Toc107845410)

[3.2 Methodology 13](#_Toc107845411)

[3.3 Requirements 13](#_Toc107845412)

[3.3.1 Must 13](#_Toc107845413)

[3.3.2 Should 13](#_Toc107845414)

[3.3.3 Could 13](#_Toc107845415)

[3.3.4 Would 13](#_Toc107845416)

[3.4 Potential Solutions 14](#_Toc107845417)

[3.5 Tools and Techniques 14](#_Toc107845418)

[3.6 Legal, Social, and Ethical Issues 15](#_Toc107845419)

[3.6.1 First Sub Section 15](#_Toc107845420)

[3.6.2 Second Sub Section 15](#_Toc107845421)

[3.7 Another Section 15](#_Toc107845422)

[3.8 Summary 16](#_Toc107845423)

[4 Design 17](#_Toc107845424)

[4.1 Introduction 17](#_Toc107845425)

[4.2 High-Level Design 17](#_Toc107845426)

[4.2.1 Tools and Syntax 17](#_Toc107845427)

[4.3 Low-Level Design 19](#_Toc107845428)

[4.3.1 Syntax 19](#_Toc107845429)

[4.3.2 Logic 19](#_Toc107845430)

[4.3.3 Performance vs. Universal solution 19](#_Toc107845431)

[4.4 Binary – The Bridge 19](#_Toc107845432)

[4.4.1 Where Software Meets Hardware 19](#_Toc107845433)

[4.5 Summary 20](#_Toc107845434)

[5 Implementation 21](#_Toc107845435)

[5.1 Introduction 21](#_Toc107845436)

[5.2 Section 21](#_Toc107845437)

[5.2.1 Subsection One 21](#_Toc107845438)

[5.2.2 Subsection Two 21](#_Toc107845439)

[5.3 Section 21](#_Toc107845440)

[5.4 Summary 22](#_Toc107845441)

[6 Testing 23](#_Toc107845442)

[6.1 Introduction 23](#_Toc107845443)

[6.2 Functional Testing 23](#_Toc107845444)

[6.3 Non-Functional Testing 23](#_Toc107845445)

[6.4 User Testing 23](#_Toc107845446)

[6.5 Evaluation 23](#_Toc107845447)

[6.6 Summary 23](#_Toc107845448)

[7 Evaluation, Conclusions and Future Work 24](#_Toc107845449)

[7.1 Project Objectives 24](#_Toc107845450)

[7.2 Self-Evaluation 24](#_Toc107845451)

[7.3 Project Evaluation 24](#_Toc107845452)

[7.4 Applicability of Findings to the Commercial World 24](#_Toc107845453)

[7.5 Conclusions 24](#_Toc107845454)

[7.6 Future Work 24](#_Toc107845455)

[References 25](#_Toc107845456)

[Appendix 1 – Project Proposal 27](#_Toc107845457)

[Appendix 2 – Technical Plan 28](#_Toc107845458)

[Appendix 3 – Title of Appendix 29](#_Toc107845459)

# List of Figures

Similarly, you can automatically generate a list of ‘Figures’. Right-click a figure (e.g. image or diagram) and add a caption labelled ‘Figure’ and ‘below selected item’. To update this after revisions, right-click in this table and choose Update Field (or use F9) and then choose to update the entire table. Delete this paragraph before submission.

[Figure 1 - Highly Technical Diagram 3](#_Toc55206440)

# List of Tables

Similarly, you can automatically generate a list of ‘Tables’. Select a table, right-click it and add a caption labelled ‘Table’ and ‘above selected item’. To update this after revisions, right-click in this table and choose Update Field (or use F9) and then choose to update the entire table. Delete this paragraph before submission.

[Table 1 - Test Results 8](#_Toc55206437)

# List of Listings

This list only applies to you if you use code snippets in your report. If you don’t have any listings, remove this whole section including the heading ‘List of Listings’.

You can automatically generate a list of ‘Listings’. After formatting your Code, move the cursor to the first line below your code block and click ‘References’ -> ‘Insert Caption’ in the ribbon menu. Select the label ‘Listing’ or add a new Label called ‘Listing’ if it does not yet exist. To update this after revisions, right-click the table and choose ‘Update Field’ (or use F9) and then choose to update the entire table. Delete this paragraph before submission.

[Listing 1 - [Program.cs] The main class of the program 6](#_Toc55206433)

# Introduction

I intend to create a programming compiler for my bespoke custom-designed 8-Bit CPU. I have been working on creating a CPU for several years, and I have now reached a stage where I can begin thinking about the software.

I am going to use Java to create the compiler, this has been proven possible in an implementation by Strom and Aas (Strom & Aas, 2001). This is because it allows anyone to use the compiler if the project was to be rolled out, as java is compiled on a VM, so it can run on any existing architecture (e.g. Windows, and iOS).

The point of a compiler is to convert high-level (human-readable) code, into machine code or binary. These binaries can then be loaded onto the EEPROM (electrically erasable programmable read-only memory) on the CPU and executed. I will be creating my language with its own syntax, structures, garbage control, etc.

Due to my prior experience with higher and lower-level programming, I will attempt to create this language with the high-level user in mind, by this, I mean I will base my syntax more on languages like Java and PHP, even though it is a direct link to the machine code like C or C++ is. By doing this, it will allow anyone to code their own program, and I will make it easy to see exactly what is going on. There will be nothing happening in the background, and everything will be visible to the user, from the interpretation to the binary output, and potentially further options to do with garbage control constraints, and manual addressing.

The issue surrounding the lack of more acute compilers is addressed in (Tanenbaum et al., 1983) which was also a motivator towards my project

# Background and Related Work

## Introduction

This chapter explores the literature that has been written surrounding the topic of low-level programming. Three themes relevant to this project will be discussed: the ways in which programming should be taught; an overview of the concepts and techniques within the low-level programming domain; and the application of microcontrollers. In addition, this chapter considers how this existing knowledge and literature overall will have an impact on this project, and to get the most out of the time and resources available.

## Teaching Low Level Programming

### Why Teach Programming

It is well known that computer programming is a difficult thing to do and is not simply one subject. It is incredibly broad and has many transferable skills. This is because it forces people to think from a point of view where they have no previous knowledge, and with a completely literal mindset. One example of this is multiplication. This is a very simple task for a human to achieve, but a computer cannot perform multiplication natively. Instead, the computer must add numbers a given number of times. For example, consider the equation:

This is a simple operation, but it forces the programmer to analyse everything in absolute depth. Some of these transferable skills will undoubtedly be reasoning skills, problem-solving, and self-efficacy in mathematics, and this is exactly what was investigated by Psycharis and Kallia (2017) in their paper on the effects of computer programming on high school students. They tested this by having two groups of 33, the control group being taught ‘Development of Applications in Computer-Programming Environments’, and the experiment group being taught Chemistry and Electrology. In addition, both groups had the same mathematics and physics courses in common. They had the same material delivered, and by the same teacher. The results of the study showed a significantly higher level of self-efficacy in mathematics, problem-solving, and reasoning skills, but unfortunately, no discernible difference was found in the performances across the two groups. Although, it should be noted that there was a higher mean score in the group with the computing integrated into their syllabus. As such, while the study was categorised as inconclusive, it could be argued that there is sufficient evidence of notable improvement in mathematics and physics from the learning involved with computer science. This elevates the importance of this project in teaching people low-level computing in a new way, as it is shown to have a positive effect on other subjects. This is excellent motivation to add as much as possible into the learning aspect of this project and ensure that the problem-solving and reasoning skills are being addressed for the impact of the project to be its greatest. Further investigation into the methods of teaching low-level programming will therefore be valuable in the design phase of this project, to ensure that the educational materials included are linked back to broader topics. This will help users of the project expand their knowledge beyond simply how the computer operates.

### Best Ways to Teach

Low-level programming is widely discounted when it comes to teaching computer programming, although sometimes for good reason. This could perhaps be the way in which it is delivered and the response to the material. Certain approaches such as a full understanding of the tools needed and full demonstrations are mostly forgotten about. Gries (1974) illustrates this point with an example case of cabinet making. In his scenario, he describes an instructor letting students use each of the tools required for cabinet making for a few minutes in isolation. While the students are now knowledgeable in all of the component skills they need, none of the students could be expected to apply this knowledge to fully constructing a cabinet on their own. This is a great example of the parallels between software and working on physical materials. It is important to note that in programming, we are presented with tools, and if not used correctly, can waste time, and cause vast amounts of frustration. Gries continues to describe ways in which to teach programming correctly, but the main points to take his paper are to: Understand the problem, devise a plan, carry out the plan, and look back. With these steps, one can “develop a level of discipline which most programmers don’t have” and transfer these skills to other aspects of one’s life, such as reasoning skills, problem-solving, and self-efficacy in mathematics Psycharis and Kallia (2017). The list of best practices to follow set out by Gries is a great way to break down a problem, and I will very much look to replicate and encourage this when developing my learning materials and programming language. I will also use this as motivation to fully teach every tool that I am giving to the user of the product. The beauty of low-level programming is that each command does one thing, and it is building these commands into layers and layers of functionality that allows users to accomplish amazing and complex things. This is the joy I wish to be able to communicate to the user of the project and to be able to facilitate this learning to any ability.

Furthermore, while providing the correct foundation to learning about the tools used in programming, it is also hugely important to cater to the experience level of the user. An advanced programmer who has programmed their whole life will not be interested in declaring variables, or how to use a for loop; in contrast, a novice programmer will become overwhelmed and put off if they are faced with advanced object orientation or nested recursion. Dreyfus et al. (1987) state that there are 5 steps from novice to expert: novice, advanced beginner, competent, proficient, and expert. It is imperative that each of these stages are being catered for. The journal goes on to explain in much more detail about the transition from being a novice programmer, all the way to becoming an expert, and most importantly, how the thought process changes over this time. Thought process is a huge factor in my project, as all I want to do is enable people to fully understand what is going on behind the screen in front of them. This again is where the use of low-level code comes into its own. As it is so close to the hardware it is being run on, it is not a case of “how to declare a variable” it is more, “how the variable is being declared”. By rearranging this initial question, we are able to create a new base level for anyone in the demographic to begin or move ahead if they already know. As such, the one material can cater to all stages. This way of thinking on my behalf will be reflected onto the user of my project, by fully understanding the material they are reading. I must, however, carefully write the documentation to enable anyone to understand and have fun with.

Another consideration is to look at a mental model first approach. This is whereby the actual programming comes very much second to the thought process behind what is going on. Robins et al. (2003) discuss that "programs are usually written for a purpose and that a mental model of this problem domain must precede any attempt to write an appropriate program". This is also confirmed by Deek et al. (1999) who describe a first-year computer science course based on a problem-solving model, where language features are introduced only in the context of the students’ solutions to specific problems. A study was undertaken to test this theory of using a mental model before starting to write any code, by Xinogalos (2014), the results of which showed that 40% of the 50 students involved in the voluntary study answer “very much” to “How much did the use of a pseudo-language help you in your introduction to programming?”. This mental model approach, whereby the students must first gain an understanding, not only of the problem they need to solve, but also to apply the limitations present in the tools they have available to them is great. Having this plan, and allowing the use of programming based solely on the mental model they have built is a great way to allow the students to think about how the computer is thinking. This also backs up Gries (1974) who states some rules to software development: Understand the problem, devise a plan, carry out the plan, and look back. This mental model approach incorporated the initial understanding of the problem and devising a plan, in one. It forces the linear approach to abide by the rules set out by Gries. As stated, these rules laid out by Gries are ones that I wish to reflect and implement into my project. I am looking to encourage the use of Deek’s mental model approach when thinking of low-level concepts, as dry running the programs in your head is an incredibly useful way to generate a simple layer of logic that is to be built upon throughout the implementation of the problem’s solution.

### A practical approach

Low-level programming is all around us, and despite this, is widely forgotten about. It is the only way in which we interact with a computer, and it is the fastest way to run any program on a computer. Low-level programming is a dying art. This is mainly due to high-level languages and structures taking the leading role in education and software development. They are more tailored to how people think and so are much simpler to pick up. Despite this, low-level programming is still incredibly relevant, now more than ever, and should not take a back seat. It is not until late education that we are taught about high-level language being translated into something which can be understood by the computer being used. In a study undertaken by Smith and Webb (2000), they looked at how meaningful learning requires pre-existing knowledge to build upon. As we are not taught how the computer actually operates when the topic of low-level programming emerges, the learner is forced to memorise information without proper understanding, and this will leave one’s mind very quickly. Their solution to this was to create a glass box interpreter based upon C, which would show a novice user exactly how the compilation process is undertaken. This is a strong parallel to my project, as I too wish to create this glass box model of an interpreter. The difference is that my glass box will be based on a custom programming language, as opposed to an existing language such as C. I think that I am like-minded to these researchers, and am out to solve the same problem. Their journal goes on to explain a great deal about the functionality of their glass box implementation and I am going to investigate drawing parallels to these functionalities, this is because the results of the study when introduced to real students was very positive, and they are features which I believe will improve my current design greatly. I spoke about some of the features in prior documentation, but this will add context and proof that this idea will work and will yield results. While the results from this study have been overwhelmingly positive, the way in which the responses were gathered was via a multiple-choice questionnaire. While these are quick and easy to set up and get results fast, there are a few issues with this form of feedback gathering; some people may simply guess the answers and still come out with a majority of correct answers.

In contrast to having a practical looking into the software side of things, one of the advantages of low-level programming is that is very close to the hardware that the code is being run on. This gives the opportunity to see the results of the code written, very easily. Microcontrollers (or microcontrollers) can have code embedded directly onto them and be used to create advanced circuitry that can be fun to make and see the results immediately. In recent years, this technology has become extremely affordable and easy to use in a DIY format. Modern embedded C compilers employ built-in features for keeping programs short and manageable and, hence, speeding up the development process Bolanakis (2017) The difficulty here, however, is that these built-in functions will not be universal, and very specific to the hardware being used, while the language I am using is extremely specific to my project, I wish to draw as many parallels to other microcontrollers, so the skills will be transferable. This approach is called Code More to Learn Even More and directs the reader toward a low-level accessibility of the microcontroller device Bolanakis (2017). This “motto” describes the way in which low-level code (certainly for educational purposes) should be written. This way, the user will have a full understanding of what is going on. The code that is actually written is incredibly simple and can act as the pre-existing knowledge that is built upon, as described by Smith and Webb (2000). It is what the user will do with this foundation knowledge that is important, as with low-level programming, there is very rarely one approach to a problem. This forces the user to really think about the problem, and once it is broken down into small enough pieces, it is completely manageable. This is the mindset that I want to be able to bring across in my project, if anyone using my language can begin to think in that way, then I will consider it a success.

## Low-Level Programming

### Low-Level vs. High-Level

Low-level and high-level programming languages are very much different kettles of fish and have very little in common. As such, it is impossible to say which is “better”, nor is this the goal of this section. With that said, there are pros and cons to both and different uses of the types of programming. These are the differences that this section is out to explore. Frampton et al. (2009) look to demystify the magic involved with programming. They state that “The power of high-level languages lies in their abstraction over hardware and software complexity, leading to greater security, better reliability, and lower development costs” This is an incredibly true statement meaning that the low-level programming languages have direct control over the hardware that is connected to a given system, and the power of the high-level languages are all based off utilising this power. The issue is that this power in abstraction removes the willingness of the compilation process to be seen by the programmer. This is shown by their later statement “opaque abstractions are often show-stoppers for systems programmers, forcing them to either break the abstraction or more often, simply give up and use a different language”. Here, they are explaining that as high-level programming is built upon low-level instructions, only commands that are deemed of use by the people making the high-level language, can be used by the developer. In some situations, the developer will need to do things that are simply impossible with a given programming language. This may force the developer to either break the abstraction (either by editing the high-level language or by writing raw assembly) in order to achieve what they need to. This is incredibly time-consuming, and expensive due to the specialism required to achieve this. Failing that, in some cases, the programming language does not and will never be able to do the operations required of it, and an entirely different language will be required to be used. The opacity is something that this project is out to address. The interpreter needs to be able to be completely transparent and explain each of the decisions being made. This is an essential skill for people who want to become specialised in this field.

In contrast, while the use of low-level programming is incredibly powerful, and can enable the developer to do almost anything with the hardware in the system, it is also the most difficult type of programming to do. This is the reason high-level programming was created: to make programming easier for humans, while also enabling computers to execute the instructions being given to it. This problem has been recognised and was addressed by Métrailler and Mudry (2015) who recognised the advent of off-the-shelf programmable embedded systems such as Arduino, which all are aimed at educating people how to use hardware, but not necessarily low-level programming. As such, some limitations exist, and this is something that Métrailler and Mudry look to rectify by building a block-based programming language being compiled directly to incredibly low-level assembly/machine code. This eliminates the need to learn a language such as C/C++ which contains extremely difficult concepts that the young demographic of these products will be spooked by and feel put off. This block-based programming was made popular by the Scratch Foundation (n.d.) who created Scratch, which is a coding language with a simple visual interface that allows young people to create digital stories, games, and animations. It is very intuitive and teaches people the basics of programming while doing something like creating a game, or animation, which (for its demographic) is a fun activity. While with this project, it is not realistic to have a visual interface, the idea of using something familiar to teach another concept is something that will very much be taken advantage of. The compile will accept the custom language which will be based on the syntax from a high-level language. In addition, the demographic of this project is someone with some programming experience, and it is this audience the language will be designed for.

### Digital Logic

Digital logic is the home of all low-level programming. It is common knowledge that there are billions of transistors inside even a typical mobile phone, but the average person is not really sure what this actually means. Digital logic is a form of thinking about how electrical signals are handled by a circuit. A common implementation of this is the TTL chip which stands for Transistor-Transistor Logic. Meaning that if you open up the chip, you will find these billions of transistors. If these are added together, you can form logic gates, such as AND, OR, XOR, etc. and by grouping these together, amazing constructs such as static memory, decision making, and mathematics can be performed. It is true that all a computer is is the abstraction of smaller things to create bigger and better machinery. Half of this project is all about creating a simplified TTL architecture to run commands on. This is a similar concept to what Rodriguez (1994) has investigated with his very oversimplified architecture. This too is very much an educational tool, and the results of which had very positive results, and due to its simple construction, was cheap and able to be used by any student that wanted it. Rodriguez did show some of the issues from his study, however. The main areas of negativity are the complexity of assembling the board, and general poor implementation. Apart from this, it overall shows a great result and encourages true promise to the project, which draws parallels to Rodriguez’s designs in that the project will also be cheap in construction, but will also learn from the mistakes made here, and focus on being simple to pick up and use.

### Relevance of Low-Level Programming

Low-level programming is still used a great deal in the modern-day, it is just a lot more specific than other higher-level languages which are easier to learn, and so more people are able to enter the industry with this experience. Some applications, however, will still require the experience of low-level programmers to create, from operating systems to general system architecture. While it is the norm to look at languages such as C++ and C# to create these types of applications, some hardware will still require intervention from languages like assembly to operate efficiently and correctly. The mere existence of recent books in assembly is a clue to this. Examples include Modern X86 Assembly Language Programming, Kusswurm (2014) and Assembly Language Step-by-Step: Programming with Linux, Duntemann (2009). These books were created to teach modern applications of assembly language, and most importantly, on modern machines, and architectures. Kusswurm’s book is actually written for an x86 architecture, which is the main architecture for 32-bit based operating systems (the name is from the 32 bit Intel 8086 microprocessor). These books are incredibly good reads, with a great deal of information about the usage of assembly in the modern-day. They are primarily based on learning assembly (not to a young audience), however, some of the techniques here will be perfect draw parallels to for this project. Furthermore, it would be pointless to make a completely new version of the assembly syntax, as such, this project will not look to reinvent the wheel, but to replicate the basic syntax, but simplified slightly for the target demographic.

### Applications of Low-Level Programming

Any computer or device uses low-level programming, after all, low-level programming is not a language in itself, it is more of a concept. Machine code is the only language that computers can use. Take Windows 10 as an example. While it is unfair to call it one piece of software, it is in fact layers of code all working to manage the PC’s resources. It can be thought of as a large piece of software called an operating system. It was created with the use of multiple languages, the kernel (the lowest part of the OS) was developed in almost pure C (a very low-level language). OS Today (2019) wrote about this and explained the primary use of C and assembly is for such a purpose.

More specifically, low-level languages such as C or Assembly can also be used for microcontrollers. They have the ability to directly access the hardware peripherals under the microcontroller’s control. This is because the languages allow for the actual manipulation of the architecture itself. This has advantages more than just being able to use all of the software, but it also improves the performance of the code to be run as the logic is not compiled or interpreted by another person’s software. It is your logic that is being run on the computer. When it gets to this low level, every clock cycle can add complexity and thus increase the time it takes for the program to run. LoopPerfect (2019) has a great explanation of this, the outline of which is that the reason higher-level programming is less performant is that generic low-level commands are required to be able to compile any generic program. Microcontrollers are a fantastic tool and are a very common use of low-level programming which this project aims to help facilitate the learning of.

## Microcontrollers

### Understanding

A picture containing graphical user interface

Description automatically generated The point of microcontrollers is not to be a fully universal tool for every application, but to be incredibly specific to the task at hand. A microcontroller used in a vending machine should not be expected to have some software changes and run in a NASA satellite. As such, it is imperative that the correct controller and hardware are chosen for the job. The perks of having such a specific system are that they will be cheap to create and manufacture in bulk, meaning that the technology can be more affordable and not have unused features. For example, if you buy the biggest and most expensive multitool out there, but you only want to open the occasional package, you won't use 90% of the other tools in its arsenal. It will be too expensive and large for the job it needs, instead, purchase a single retractable knife, it performs one job, it is cheap to purchase, and gets the job done just as well as the expensive one, if not better. A microcontroller is the knife in this example, a cheap and cheerful solution, and with a few subtle tweaks and additional hardware, can be the solution to any problem to any problem. The ability to chose which controller to use, and which hardware to use as the peripherals is the difficult part, and much research should be put into selecting the right tools for the job, as with any solution. Yılmaz et al. (2017) wrote an article to explain exactly this. They do this primarily by comparing various different types of controllers, peripherals, and architectures. The most interesting and relevant to this project however was the comparison of development boards:

Figure

These are great boards, mainly for educational purposes, but can also have industrial applications. While some of these boards are very well known and used a lot in education, they do very much hide a lot of the ins and out of what they are actually doing, but such is the nature of higher-level architectures. The raspberry pi, for example, is not a microcontroller, as it has a full operating system, “a computer” is a much better description of what it is. The wide range of technologies available in this area is made very obvious from this study. While my project is not aimed at advising the user about which microcontrollers and hardware to use, I do feel as though understanding the limitations given by low-level programming will enable them to make their own decisions and be able to carry out their own research as to what is available to them to solve the problem they are out to face.

### Applications

Microcontrollers are processors which have the capability to be used on a wide range of devices, from calculators to vending machines. While such devices like this on the outside are very much a hardware product, they would be completely useless if it were not for the software burned into the microcontroller itself. This is demonstrated by Tanshi and Bello (2014) who show the vast amounts of applications of the µController. The article is mainly focused on the implementation of a basic calculator using a µController as the core. As such, a big focus of the paper was about the hardware requirements for such a device, and this is where the power and flexibility of microcontrollers come into play. The µController is only as powerful as the peripheral devices it has available to it. Here, Tanshi and Bello are using hardware to create a user interface for a calculator, but could just as easily make one for others such as automatic transmission cars, fuels pump meters, washing machines, digital cameras, DVD’s, mobile phones, vending machines, etc. Just from this, it is clear that microcontrollers are all around us, and they will only ever become more powerful, flexible, and most importantly, still in demand. This demand drives up the need for people who are able to program such devices in the future. People who are used to only programming in a high-level environment will simply not have the know-how to make devices like the ones listed by Tanishi and Bello operate. One might argue that modern microcontrollers should have the ability to use high-level techniques, but this adds complexity, and cost. This thus size to the units themselves and begins to degrade the very point of having such a low level and specific device. The goal of this project is to explain to users how low-level programming can be achieved, thus helping to develop the next generation of low-level programmers and to spark interest in this dying art.

In addition to consumer products being powered by microcontrollers, research projects have been undertaken by countless academics and companies to better their products or research with the use of microcontrollers. This is due to the relevance, availability, and simplistic nature of the microcontroller which shows a true reflection on how good the technology is. Shao (2006) created an Improved Microcontroller-Based Sensorless Brushless DC (BLDC) Motor Drive for Automotive Applications. He developed a cheap and effective new way to monitor the back EMF produced by electric motors which can more accurately monitor the motor positioning, while also providing overcurrent protection. Zwerg et al. (2011) created a new form of microcontrollers for energy harvesting applications. It draws only 0.000082Amps (82µA) which makes this technology perfect for applications in energy harvesting where every nano amp drawn, counts. Babiuch et al. (2019) Even take a look at the applications of the internet of things with embedded technology, which is something that everyone will soon be using, and most of the time without the user's knowledge. As shown, there are countless applications for this type of technology, which are aiding in the progression of many different industries that are used by all types of people all over the world. The applications will only continue to grow with the development of new technologies and industries that we don’t even know about today. As such, it is imperative that people remain clued up in this area to keep the up-and-coming technology developing. This project is aimed to achieve this education upkeep, as low-level programmers are becoming few and far between with high-level technologies being easier to understand.

### Relevance

With the knowledge of what microcontrollers are, and how old the technology is, one may begin to wonder what the point of learning this old form of thinking is. With great advancements in technology, surely the use of these small and specific devices will phase out, and no one will need to know any low-level programming languages or logic. Mitchell (2020) made an argument for this being far from actual fact. His primary arguments consisted of the incredible complexity of larger, more powerful microcontrollers, along with the vast increase in price which is attached to this. He concludes by stating that the industry shows no sign of a reduction in 8-bit usage, and their low-cost nature combined with simplicity still makes them highly relevant. In addition, Pal (2017) covers some very interesting information about the reasons for using an assembly language (a very low-level type of programming). He shows how the use of assembly offers complete control over the system’s resources, direct access to hardware, a deep understanding into how the computer actually operates, complete transparency, and more. These points are extremely potent in this project, as it aims to address each of these positives for learning low-level programming. The whole point of the project is to be as transparent as possible while also providing a somewhat powerful outcome. All of the hardware will be available all of the time, meaning the user can fully grasp what everything does and will have the ability to play around with all of the parts. As shown, this technology is still rife in the modern world, despite it being old. This gives me great confidence in the project and only strengthens the fact that this is needed and there is a serious gap in the market here.

### Uses in Education

The point that microcontrollers are not taught very much in education due to their complexity, is a major driving factor of this project. “Microcontrollers account for the majority of processors produced today, yet their capabilities are seldom explored in modern computer science curriculum” This is a sad quote from a study by Margush (2006) which goes into the importance and relevance of assembly language. This quote is one thing that my project intends to address. The study by Margush continues to talk about different ways in which to compile assembly code. One example of this is for the Intel 8080 microcontroller “Windows XP platforms to write, assemble, debug, and execute Intel 8088 assembly language programs”. The issues that are raised here regard the way in which the OS allows access to low-level things like I/O and interrupts This is a theme that is shown across a wide range of assemblers (the compilers for assembly code). This problem with transparency is again a problem that prevents people from gaining a further understanding as to what goes on inside the computer. The usual phrase used is that “the code is turned into something the computer will understand”. The project will ensure that there will be complete transparency in the workings of the compiler, showing the user the output of stages as it goes and shows the final output, so the user has a complete view as to what is going on.

## Summary

This section discusses the current literature surrounding the area of education and applications of low-level programming and how the lessons learned here can be applied to this project.

The education of low-level programming has become rather limited and is barely taught at all due to the invention of high-level languages. While this is not necessarily a bad thing, it does introduce some potential limitations for advanced developers who have reached the extent of the functionality of high-level programming when it comes to fully utilising the resource of a PC; Integrating hardware into their projects; or simply executing an operation which is not possible given the constraints on the high-level programming due to its very nature of abstraction. It has been shown that there are great benefits of teaching programming and starting low-level principles will improve anyone’s literal and logical thinking which is incredibly transferable to other STEM-based subjects.

The application and usage of microcontrollers is also a primary talking point of this chapter. Their relevance in the modern day is still very much evident with almost every device in the world running some form of microcontroller with low-level software written on it. As such, the need for people who understand this type of programming will only increase. The performance of the low-level direct hardware addressing is a large driving factor for this. With hardware, for example, the process cannot simply be paused, the controller must be available to process any input it provides, or critical failure could occur.

There is most definitely a place in this world for high-level programming, and this chapter proves this, but it also analyses the other side of the argument whereby low-level languages should not be forgotten about and thought of as simply another layer of abstraction that is too complicated for anyone to understand. This would be incredibly sad, as the future of computing would be much less performant, and it would be near impossibly to acquire full implementation of the latest and greatest in peripheral technology.

To finalise, both forms of programming have a place in this world, but respect should be given to both, and not to forget about the very foundation of modern-day computing, as without it, even today, almost every modern commodity would not exist.

# Project Planning

## Introduction

This chapters takes a look into the approach taken for the project along with requirements that are needed to be hit in order to consider the project an overall success.

In addition, some of the first and most basic decisions will made with regard to the approach of implantation and justification as to why I have chosen them. Furthermore, I will also consider some of the legal and ethical issues surrounding the project.

## Methodology

This project will be worked on only by myself, and as such, fully utilising the agile approach is not an option, as this is aimed more at team working and management with a customer [REFERENCE AGILE manifesto] With that said however, I am looking to use some elements of the agile way of working such as the MoSCoW approach of requirement gathering, and time estimates which allows for frequent re-evaluations of progress to avoid the quality of the overall project being the area which takes the a hit in quality if time is becoming a major constraint.

## Requirements

My goal is simply to create a programming language which acts as a tool to make my custom built 8-Bit computer into a Turning Complete Machine [REFERENCE]. As such, my primary requirements will be to provide the user with enough tools to enable any solution to be created for any problem. This is achieved by using the 3 basic principles of programming: sequences (instructions), selections (if statements), and loops (while, for loops etc.). With these tools, in theory, any solution can be created to solve any problem [REFERNECE].

Using the above mentioned MoSCoW technique, I will set out my requirements:

### Must

1. Be a language that use high level syntax with parallels to existing languages to enable a shallow learning curve for new users of the language and aspiring low level software engineers
2. Enable the steps of compilation to be explored at the users own pace and thought process and for a bespoke experience every time
3. Convert the high-level language into the bespoke and simplified assembly commands, and the resulting binary which is compatible with my custom 8-Bit computer
4. Be considered at least a Turing machine capable of solving most any problem within the obvious constraints of an 8-Bit system.

### Should

1. Be a single executable for use in the user’s very own compiler every time giving them control and understanding of the steps needed to be taken in program compilation
2. Be able to provide the user with useful and correct feedback as to syntax errors when compiling the program
3. Have built in garbage control to prevent data leaks and allow for easy memory management within the language

### Could

1. Have additional custom built in functions to the language specifically for this machine

### Would

1. Be able to manage scope with variables to save memory with values which are not required outside of a given block
2. Provide stack trace debugging errors and associated debugging tools to get to the bottom of any issues the user has.

## Potential Solutions

Given the demographic of this project, languages such as C, C++ or GoLang would be obvious choice, what with their already low-level nature, and compilers even nowadays are written in these languages too [reference]. The difference here is that these languages are low level in a way that is close the hardware they are already running on (the modern computers of today). This is why low-level languages are so specific to the machine they are running on. With this constraint of being close to the wrong type of hardware being removed, the only hardware that the code needs to be close to is the hardware of the 8-Bit computer. I have no issue with utilising the modern processing power of the computers of today in order to enable the learning of hardware programming. I am therefore more inclined to use languages such as Python, Java, or C# which are very commonly used, heavily documented, and easy to get going with. At this point, my own personal preference will come into play, and Java is my preferred choice.

Not only is it the language I am most familiar and comfortable with, due to its compilation method (using a Java Virtual Machine [Reference]) it can run on nearly every modern computer, meaning that anyone will be able to run my code. In addition, Java has a great many powerful tools and features built in.

## Tools and Techniques

Java has a great many powerful tools and features built in, and I intent to take full advantage of this. My main focus is to make the compiler as modular as possible, and with the interest of meeting my first “should” requirement, the project will not be a complete compiler. The project will consist of all of the tools required to create a compiler, but it will up to the user to create it however they see fit. The primary tools that I will supply to the user are:

* Lexical Analysis
* Syntax Analysis
* Code Generation

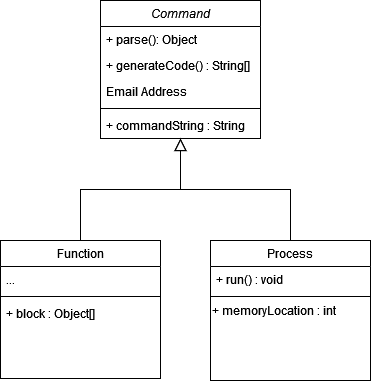
You will notice that these are also the 3 steps to most every compiler. The user will either be able to use built in debugging features within their IDE to stop the compiler running at those given stages and analyse the inputs and outputs, or will be able to call output functions within the executable and output the results directly to the console. This modularity will allow users to create their own compiler and work at their own pace to full understand what is going on in the usually shut box (compiler).

The primary tools and techniques that I wish to utilise within Java to make this achievable are:

* Object Orientation
* Recursion
* Polymorphism

I plan to keep everything in organised objects, and the hieratical ordering of the commands after the lexical analysis will be stored in arrays of these objects. Furthermore, I plan for these objects to be stored in blocks, in the same way blocks of code work in any other language, usually depicted by the open and close curly brackets “{}”. I will then use recursive techniques to parse each of the functions and commands in the blocks at the syntax analysis stage, which means it is possible to have an infinite number of blocks within blocks and so on.

One constraint of Java which is also what makes it such a good language is the use of types. One example of when these are used is in the creation of data structures like ArrayLists and Queues. If the language only allowed the user to undergo one function, this would be no problem, as there could simply be a single array of a custom object type. However, if this was the case, the language would not be good enough to enable the 8-Bit computer to be considered a Turing machine, and would not be acceptable for this project. Instead, I will have an abstract class containing a generic “Command” for example, and the master array can simply contain a list of these classes of which the function and process class can extend, and be stored in that same array to keep everything together.



Figure

The elegance of polymorphism here is evident, as both of the child classes (“function” and “process” in this plan) have a “parse” method inherited from the parent abstract class, but each one will override this parse function and do different things, although fundamentally they can both be considered a “Command”.

In order to implement these tools and techniques, a variety of existing tools must be utilised. The main tool is the IDE – the IDE of choice will be VS Code, this is a lightweight version of Visual Studio, and allows for additional features to be added if required. The idea here is that you are not forced to have anything installed – taking up storage space, and potentially slowing down the computer [REFERENCE – pros to vs code]. As Java will be the main language, the Java addon will be installed which integrates the JDK in with the main code editing area.

In addition, as a way of loading OS and program data onto ROM chips, an Arduino will be used. As such, the official Microsoft Arduino extension will be installed onto the VS Code environment.

## Legal, Social, and Ethical Issues

# Design

## Introduction

The purpose of a compiler is simply to translate one script to another lower level script which is closer to the hardware of the computer [POSSBLE REFERENCE]. As such, there are really two major areas which need to be designed: The higher level syntax that the user will interact with; and the lower level syntax and logic that will be act as instructions for the hardware to run. Both are as important aspects as each other, and they will need to be gotten right in order for this project to be a success.

The high level syntax (for gaungau) will need to be easily readable by the programmer, and should have some resemblance to existing language so it can be a pick up and go experience, rather than a pick up, read a manual, and then go.

The lower-level syntax however is where some performance techniques will need to be used that do not limit the power that I hope to offer from this tool. A balance must be struck, and a great deal of time, trial and error, and patience will be needed to develop the best solution for any given logical scenario.

This chapter takes a look at the design choices involved with the primary aspects of the language and assembly language. Although further choices will need to be made during implementation, it is important to go into a large project such as this with some plan to hit all of the requirements decided on earlier in the planning process.

## High-Level Design

### Tools and Syntax

In order for my project to be considered a success, I must meet all of my requirements, the basic ones which I base my entire project around is to create the functionality in my programming language to allow the user to use sequences, selections, and loops. These are very simple, and almost every algorithm can be broken down into these key attributes.

The Backus–Naur Form (BNF) is a way of describing syntax grammars with no context, meaning that while the danger of garbage in garbage out with this notation is true, if done correctly, it will describe the different areas of my syntax for the language in a standard and know format.

#### Sequences

##### Syntax Decisions

This is simply a single command which could involve arithmetic, or manipulating memory on the computer. As such, in order to hit this criteria, I will have the ability to set variables to values and do arithmetic within the same line. An example of this in Java is:

int num1 = 5 + 9;

This will obviously simply set the memory address associated with “num1” to the sum of 5 and 9 (=14). While this again, is a very simple task, it is one of the fundamental things you can do with a computer.

The above example is in Java, and this is a language widely used by a great many people with syntax that is easily readable and would be a great language to base my initial design off, however, I will be making a few tweaks as the functionality of Java greatly exceeds my plans for the gaungau language. One example is that my language will only be able to work with integers, and not other types such as chars, or floating point numbers. This might be a good way in which I can improve my project in the future, but for my purposes as of now, I will focus on the use of integers. Due to this fact, I will remove the need for defining the datatype of the variable. This decision will increase the ease of use of the language, however, it might not be obvious initially that this is required, and will need to be made paramount in the documentation.

While this is all true of Java, in other languages such as python, the datatype does not need to be set when defining a variable [REFERENCE PYTHON DOCS]. A user coming from a python background might find this an easy change to make.

Another design choice I decided to make (pretty much just because I can) was to make the equals sign of setting the variable a bit different. Instead of using just the standard “=”, I am going to use “:=” instead. This is in reference to the Dafny language [DAFNY DOCS REFERENCE] which I grew to like.

As a result of all of my design choices, the syntax I am settling on is to use the following:

Num1 := 5 + 9

Notice here, there is no datatype choice, there is the different assigning syntax, but apart from that, it is most like any other language, and anyone can read the intension of this line is to set the variable “Num1” to the sum of 5 and 9 (=14). This is just one example, but covers most every possibility when it comes to variable assignment such as:

* Assigning variables to each other
* Assigning variables to calculations containing other variables
* Assigning variables to even longer calculations such as 1 + 2 + 65 + 30 – 70 – 4 + 2 + 2 + 6 – 8…

I am happy with how this syntax reflects this feature, and if implemented correctly will meet this requirement

##### BNF

<symbol> ::= <expression>

<expression> ::= <integer>

| <symbol>

| <integer> “+” <expression>

| <integer> “-” <epression>

| <symbol> “+” <expression>

| <symbol> “-“ <expression>

<symbol> ::= <charater>

| <character> <symbol>

<character> ::= <integer>

| <letter>

#### Selection

##### Syntax Decisions

In programming, selection (or branching) is the ability for the computer to make decisions based on existing conditions and values [REFERENCE what is branching]. This is by no means the computer making its own decision in some kind of AI capacity, but in a term more widely known… an if statement. If statements are incredibly widely used in most every language, and the syntax is almost identical for each language following the rule “IF<condition><block>”. I intend to use this same structure for my language and not deviate too much from this, in order for the learning curve to be as small as possible.

Furthermore, another consideration is the use of the “else” statement. One could argue that the use of the else statement is entirely possible using only if statements as seem below in the Java example:

if(num1 == 5){

    // Do something

}

if(num1 != 5){

    // Do this instead

}

As we can see, there is no else statement in use, and using num1 as 14 from the previous examples, the second if statement will be executed in the exact same way an else statement would be used. While this much is true, I can’t think of a single language that can support an if statement with that type of syntax, that cannot support an else statement. With this in mind I will be implementing an else statement in my language, and will try to keep the syntax as close to the following as possible:

If(num1 == 5){

# Do something

}else{

# Do something else

}

That is the very syntax in gaungau that I wish to implement. It is identical to other high-level languages out there, and will be quick for any new user to pick up and create solutions with.

##### BNF

<if statement> ::= “if(“ <condition> “)” <block>

<block> ::= <statement>

| “{“ <statement> “}”

| “{“ <statement> <statement> “}”

<statement> ::= <variable>

| <statement>

#### Looping

##### Syntax Decisions

Looping is the ability to loop over and repeat code but with a potentially new state than the previous run.

##### BNF

<while loop> ::= “while(“ <condition> “) <block>

<block> ::= <statement>

| “{“ <statement> “}”

| “{“ <statement> <statement> “}”

<statement> ::= <variable>

| <statement>

#### Custom Tool

The result of all of the main processes listed above, is going to be a custom-made instruction set of assembly instructions, based loosely on the syntax on an x86 system. This will in turn be converted directly into binary which can be run on the custom 8-bit computer. While this is a completely bespoke language and cannot be run on a standard modern computer of today such a windows machine, I want to enable the people who want to take their understanding one step further, to do so. The user will have full access to the outputted assembly code, and them are free to manipulate this as much as they see fit, however, free hand editing is not feasible really when it comes to sharing their code with friends, or to a classroom of fellow classmates. I want to enable full transparency, and for a powerful function to exist which can edit the very assembly code without a check but the compiler other than a basic syntax check. This function has the ability to throw off an entire program, and cause data loss, counters to clear accidentally, and literal control over every aspect of the computer. On the other hand, it will also allow someone who knows what they are doing to create more performant programs than even the compiler will be able to produce. The main reason however is simply to allow the user to play around with all of the hardware they have in front of them without constraints of error messages, or someone else telling them what to do which is fundamentally a large factor of modern-day high-level programming.

As the function will enable the alteration of the raw assembly code being produced, I will name this function “raw()” it will accept one parameter which will be the single instruction that will be inserted blindly into the code generation.

For example:

Num := 0

While(Num < 10){

Num := Num + 2

Raw(OUT) # Could equally be print(Num)

}

Here is a simple algorithm to figure out 10 / 2 however, the outputting of the number is done via the use of the physical assembly instruction “OUT”. I intent to include a basic syntax check, so the parameter "GOLDFISH” would return an error, this is only to ensure that the program is full runnable no matter of the outcome. This is a simple example, but shows how powerful it could be if used by the right person.

## Low-Level Design

### Syntax

The syntax of low-level language is not so well known [REFERENCE], and as such, there is less that I can use to base my design decisions on. That said however, while not widely used on the outside nowadays, a very common assembly language is the x86 instruction set [reference]. The premise of this is a set of (usually) three letter instructions (with a few exceptions) which are character representations of the operation code associated with the operating system to tell the computer which set of binaries to execute at a given time. The compiler will produce assembly codes following this same paradigm as it has been proven to work and will aid is reducing the learning curve into assembly languages, and assemblers like NASM [REF] and MASM [REF] which are run on windows systems.

### Logic

The low-level code was written to cater for only one function at a time. This means that each instruction is modular and can be independently debugged and

Granularity, modular, small individual tasks compared to high level stuff which does loads

### Performance vs. Universal solution

One of the main disadvantages of high-level languages is that they are very inefficient as they are so far away from the hardware [REFERENCE]. This is a consideration which must be thought over carefully. The point of this project is to create a low-level platform upon which leaning can be done, do to create the associated language with many obvious built-in inefficiencies would be counterintuitive.

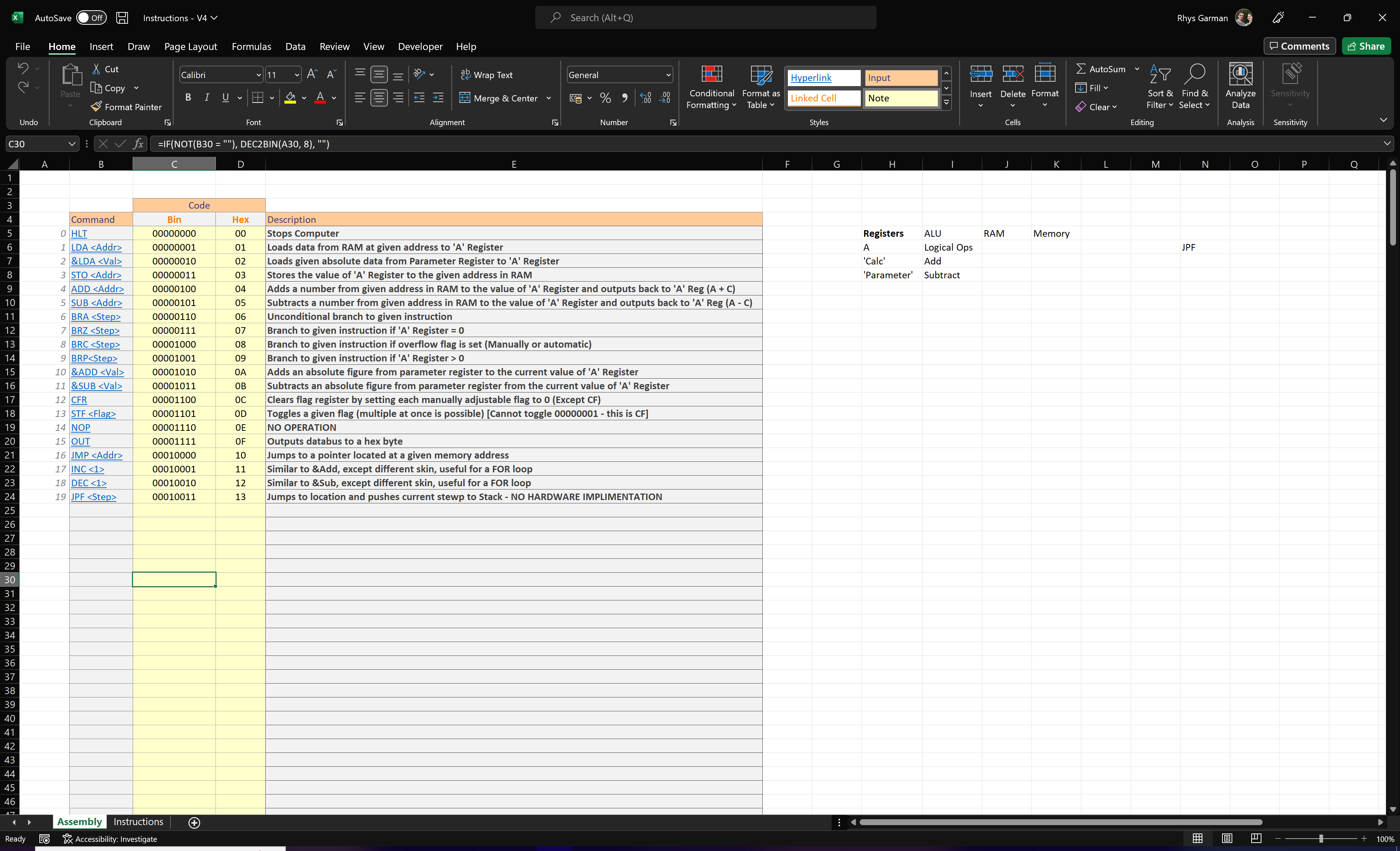
This type of problem is one which can only really be discovered when undergoing the implementation phase of the project. As such, it would be almost impossible to identify them at this stage. Saying that however, the mitigation of these issues should be considered from the off, and measures put in place not to prevent them, but to limit the use of inefficiencies.

With the language being so very close to the physical hardware of the computer, it will fundamentally be more efficient than other languages, it will however have most of the same problems faced by other languages. A huge advantage that this computer has over others, is that the hardware is designed specifically to execute this language, meaning that hardware changes could be made to make the language run more efficiently, and potentially removing the planning inefficient methods all together.

## Binary – Where Software meets Hardware

Every computer will require an operating system to run, even my little 8-Bit computer. A standard OS will be able to manage the goings on of the computer, and control programs to run as and when needed [OS definition reference], however my OS will only be able to manage the running of one program at a time. While this is very simplistic, it is all that is needed at the moment. I plan in future to create a graphics card to be able to use any VGA capable monitor to display information on, and eventually to create a WYSIWYG interface on which to create programs and trigger the execution of programs. At such a point, a more advanced operating system will be required, however, for now, the ability to manage the one program is all that will be needed.

The operating system will need to be written in a way that the computer will understand, the user will not need any knowledge of this, it will be the final interface between the user and the computer. Everything past this point will be pure binary in a format that is almost impossible to write, and most importantly debug. The binaries which make up the OS are stored on ROM chips mounted to the computer. This may seem like a complex task for a few 1’s and 0’s to accomplish, but it is the very foundation upon which everything about this project relies. Every high-level command, like an if statement, is translated into a bunch of assembly instruction, which are translated into another list of binaries which are simply symbols of electrical charge. Pins on the logic chips in the computer can detect these voltage differences, and active/deactivate upon these triggers. The charges are stored inside the ROM chips, or more specifically the EEPROM (electrically erasable programmable read only memory) chips. These are a very basic form of non-volatile memory; in that they do not lose the data when turned off. This type of memory storage was the main bottleneck of old computer systems, it was very expensive and not hugely reliable [REFERENCE]. Nowadays however, one chip which can store 8192 bytes of data costs only around £2, and is the technology used here to store the OS.



The above is the complete list of assembly operations which can be run on the computer. The command on the left is the instruction in “word” for, and is one step easier to read than the raw binary. The number value on the far left column is what I am using to assign the opcode to the instruction, and the Hex conversion is the actual opcode I will reference e.g. LDA -> 0x01 and OUT -> 0x0F.

The opcode refers to the address in the ROM which contains the list of binary instructions to be executed.

Calendar

Description automatically generated

Above is the list of binaries associated with the LDA function which takes a value from a given address in RAM and loads it into the A Register. While on face value this seems a bit overwhelming, and complicated, all of the binary is hidden behind the language, and a good experience can still be had with this learning tool even without looking at the binary. For those who want to go deeper however, has the ability to do so with the complete transparency tactic I am taking with the project on a whole.

This operating system was designed ahead of time, and again is open for modification by anyone who knows what they are doing. That is the power that this project is providing and will give users the full experience of how computing actually works.

## Summary

In this chapter, I have outlined and justified the design choices I have made when creating the syntax for the custom language. In general, the reason I have made the choices I have, is to make the learning of this new low level programming as simple as possible. As this is an educational tool, I am not trying to prove to anyone of my own skills, I instead want to others to learn this dying art of understanding exactly what is happening inside the box, and offer tools which no other language allows you to do, such as actually editing the assembly code that is being generated.

# Implementation

## Introduction

## Main Processes

### Lexical Analysis

The lexical analysis is the first and most basic step of compilation. In simple terms, this stage is all about turning the code writing in, near enough plaint text, into a format which is standard, and can be understood by the syntax analyser (the next step). My lexical analysis took a few small and basic steps to complete its analysis:

1. Have each instruction on different lines including multiple instructions per line with the use of semi colons or line breaks.
2. Remove comments
3. Moves each brace to a new line
4. Removes blank lines

An example of the code running through the lexical analysis is:

Num1 := 0; Num2 := 8

If(1 == 1){Num1 := 5}

Else{

Num2 := 5

}

Which will be converted into:

Num1 := 0

Num2 := 8

If(1 == 1)

{

Num1 := 5

}

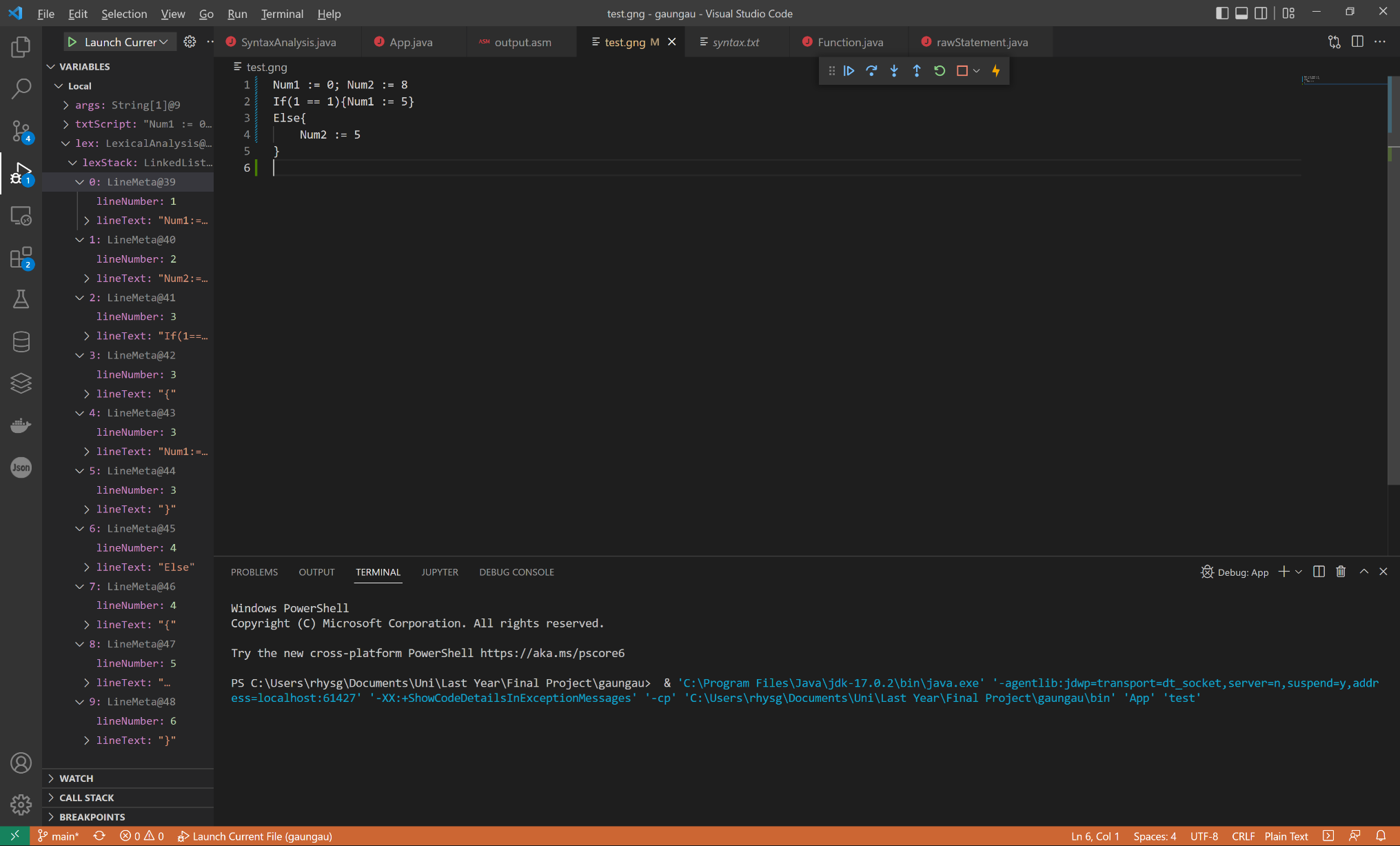
Else

{

Num2 := 5

}

I will use this example in the actual compiler, showing how the user will be able to see exactly what is going on.

We can see the code on the right entered exactly as seen above. And the lexStack in the debugger window on the left is the output of the lexical analysis.

NB: Note here, the actual ability to do this. Someone can stop the program at any point, and see exactly what is happening.

We can see that each of the elements in the lexStack contains an object called a “LineMeta” This contains the actual raw code along with the line number in the file associated with that command. This is useful for error messages, so it can figured out exactly which line on the real that is causing the issue.

We can see that each of the elements have been identified and put in its own object to be executed later.

Also note, that this language is case sensitive, so the keyword “Else” is different to the keyword “else”. At this point however, the compiler does not care. It is simply putting the code into a format which the later steps expect and can accept.

If this code was to be fully executed, there would be many syntax errors due to the upper-case keywords used throughout.

While this step is as important as any step taken to compile the code into binary, the actual code itself is fairly uninteresting, containing mainly a vast amount of string manipulation

### Syntax Analysis

The syntax analysis is where the program created is checked for any errors in syntax and logic, and the resulting valid and correct code is then converted into a hierarchical tree to show scope and blocks.

#### Phase 1 Syntax Check

Each line is checked for the presence of:

* Else statements
* Variable setting
* An open block parenthesis (“{“)
* A Close block parenthesis (“}”)
* A function
* Any keyword
* Built in functions
* Creating a new function

In that order. At this point, the compiler flags any syntax errors, or logic errors such as floating closing parenthesis. Take the following example:

x := 0

y := 1

else{

test := 9;

}

while(1 == 1){

print(y)

t := y

y := x + y

x := t

}

}

Here, there are 2 errors which should be flagged at this point, highlighted in green is an else statement with no if statement attached. This is a logic error, although the syntax is technically correct. In addition, the closing bracket highlighted in red has no matching open parenthesis, and as such will also be flagged. Helpful error messages will be displayed in the console to indicate to the user where the issue(s) are:

Syntax Error: Invalid placment of else statement at line: 5

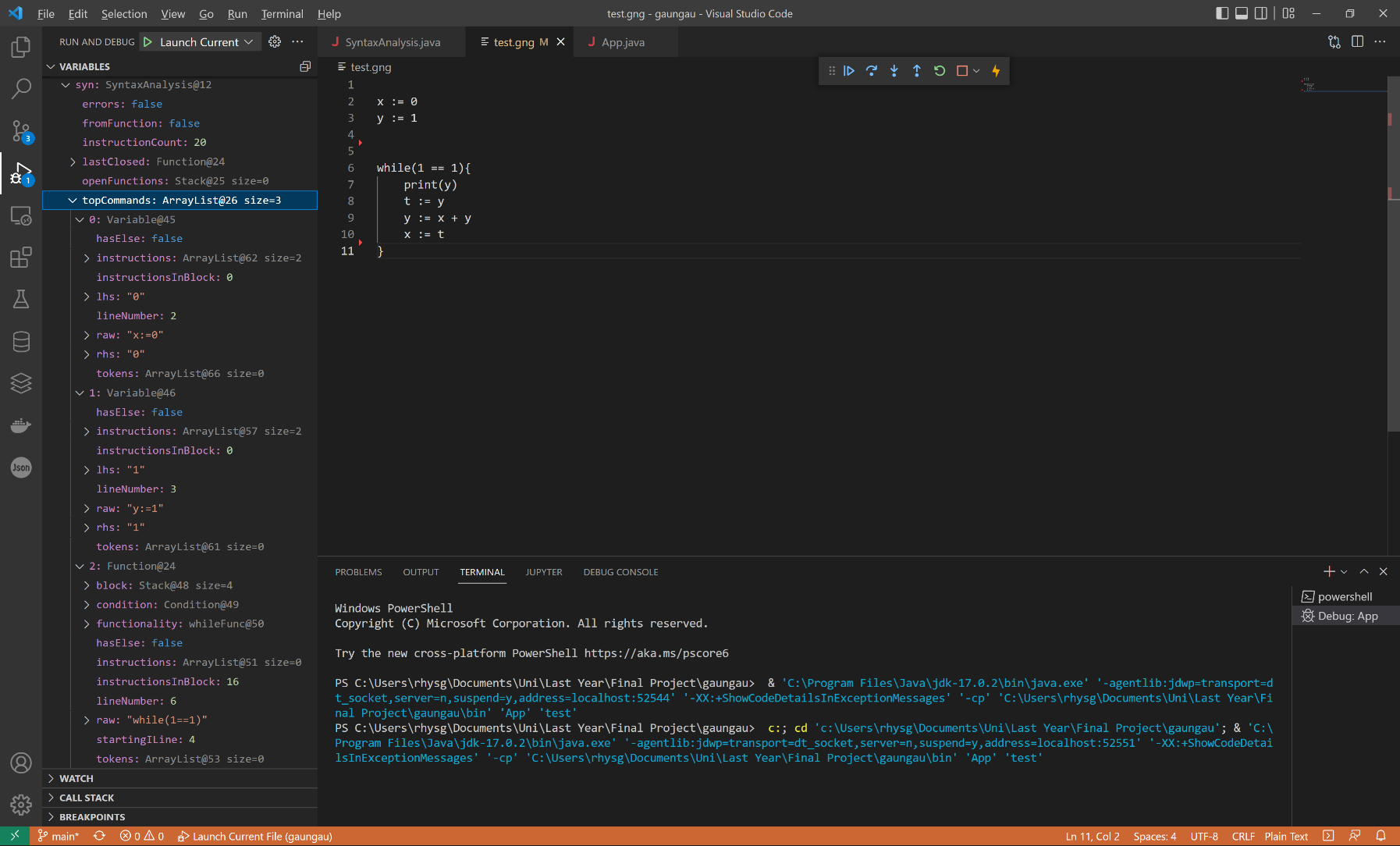
Syntax Error: Stray '}' at line: 18

#### Sorting into Blocks

After the code has been transformed into a state that is uniform and expected by the compiler, it can now take a look at sorting the code into blocks and creating the objects linked the functionality of the string representation of the function. For example, a “Variable” object will need to be created when the statement “x = 5” is written. I have approached the block problem by using a hierarchical tree to represent the code (this makes the next step easier). The general rule in order to make this happen is open a block where there is an open curly brace, and every subsequent statement will be stored inside of this block until a corresponding close brace is found, at which point subsequent statements will now either start a new block, or be placed in the parent block until there are no lines left.

With the compiler being able to be stopped at any point makes it easy to show how this has been achieved. Using the following example of the Fibonacci sequence, we can see how the compiler splits the GNG code into the tree of blocks.

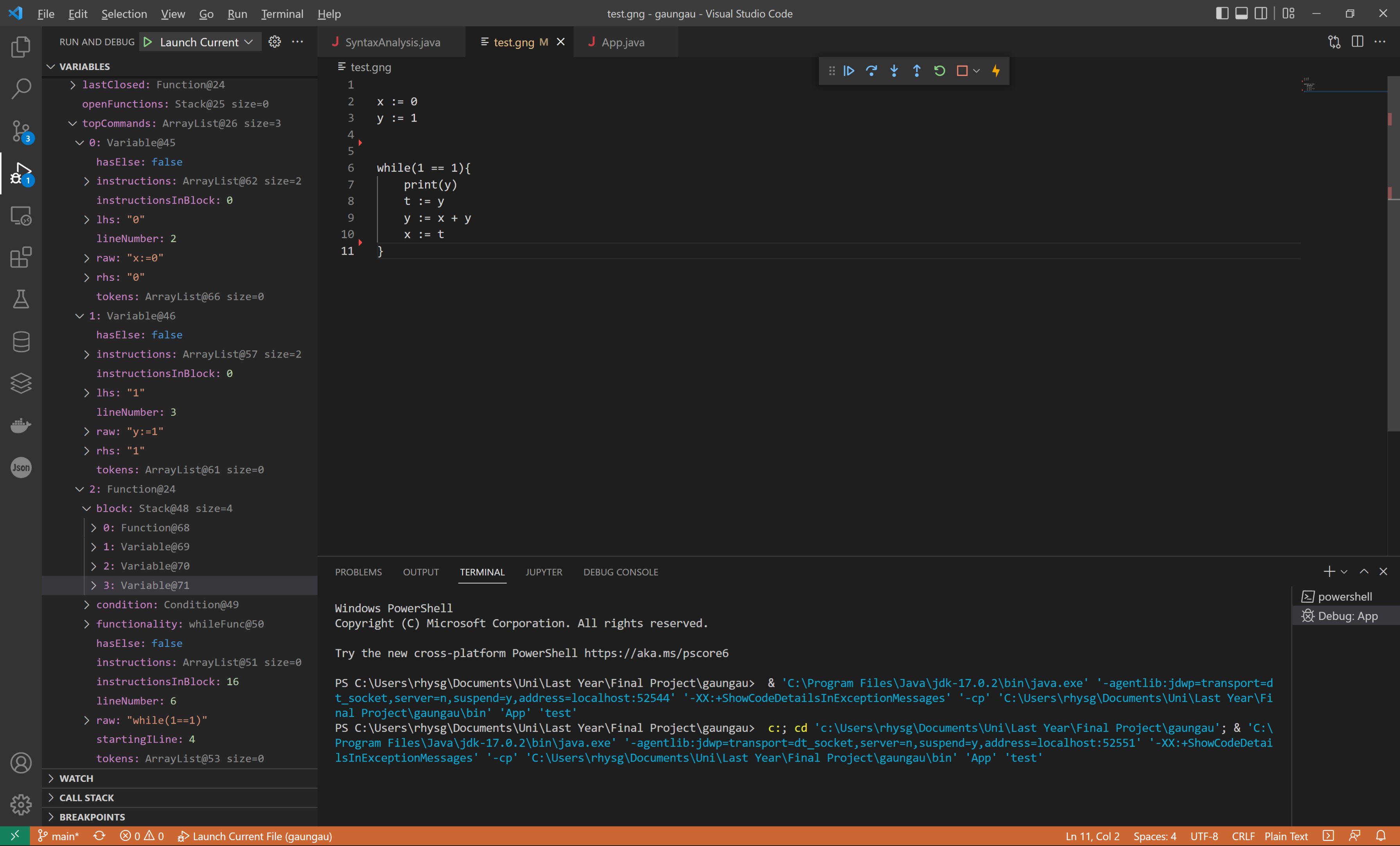
Notice that out of the 8 lines of code written in the editor to the right, only 3 commands have been identified. This is not a mistake. Note the type of the first two lines, they are both called “Variable” this is consistent with the first two lines of the program. The next command is listed as a “Function” This is because I have called While and If statements, “Functions” in the interest of polymorphism and ease of expansion later on. The function here is of course the while loop present at line 6, and this can be confirmed by looking at the “raw” property stating the content of this line. In addition, there is a property called “block”. This “block” property is the ArrayList which contains all of the subsequent commands found within this while loop block.



When the parsing stage comes around, the only initial commands that will be parsed are the ones at the top, and as such it is only these three which are parsed. The rest of the commands (inside the while block) will be parsed when the recursion occurs later.

After taking a look inside of the “block” property inside the while loop function, we see the following:

a function followed by 3 variables. This is in reference to the print statement, and the subsequent variable changes found after it.



We can now see that all of the commands have been located and stored in the correct block to reflect the structure of the code.

As previously mentioned, the syntax analysis will also convert the string representation of the intended function to the object associated with the actual functionality. Take the following example: “x = 5” will need to create a “variable” object so when the parsing and code generation comes around, the compiler will know which assembly instructions to be executed. There is rather a large if statement to catch all the differences in syntax to correctly identify the functionality which objects are needed to be created. Using the above variable example, the analysis will seek out variable syntax, in this case is “:=”.

// ======= VARIABLE ======= \\

if (statementText.indexOf(SyntaxCfg.variableOperand) > -1) { // Identifies a variable based on the presence of ':='

if (openFunctions.size() > 0) {

    openFuncAddStatement(new Variable(lineMeta));

    } else

    addTopCommand(new Variable(lineMeta));

The “variableOperand” reference on the second line is just a reference to a constant containing the actual identifier (“:=”). The current line being checked (statementText) is checked for the presence of this variable identifier with the use of the “indexOf” function. That identifier is the unique phrase used when manipulating data in RAM (altering or creating a variable) and as such, can safely assume that this line is related to variables, so the corresponding object is created and added to the tree.

#### Parsing and Phase 2 Syntax Check

The parsing process converts the existing tree (which now contains functionality objects) and parses them. The result of the parsing will be the actual assembly instructions of valid syntax. Any issues at this point will be flagged with useful messages as to how the user may go about fixing the problem. This may sound like a complex and long process, I have made good use of recursion to parse each element in the tree in order and without special changes for each statement.

// ===== Parsing ===== \\

if (!errors) {

// All the code at this point makes sense, but not necesserily correct \\

  for (int i = 0; i < topCommands.size(); i++) {

    if (!setError(!topCommands.get(i).parse(instructionCount)))

            instructionCount += topCommands.get(i).instructionCount();

            if (errors) break;

        }

    }

}

The above code snippet is all of the code for triggering the parsing of each object. While this is technically true on the outside, a huge amount of processing takes place when the parsing is triggered. The parse function also contains polymorphic behaviours, as the “parse” function can be called on variable object, and function objects, as they both extend the “Struc” class, and override the parse function to have the parsing functionality expected of the given structure. For example, parsing a function will need to check for keywords, and built-in functions before creating the attached functionality, whereas the variable object will need to perform several name validation, and RHS validation to ensure that correct values have been passed in.

The recursion occurs when blocks are present in the code. In the Fibonacci sequence example above, there are only 3 objects on top to be parsing by the main parsing loop, but there are more commands which need parsing inside of that while loop.

for (Struc structure: block) {

if (!structure.parse(crntILine))

        return false;

            crntILine += structure.instructionCount();

}

This is the loop which will trigger the parsing of the instructions inside of a block, and if there is a block inside of a block like:

If(valid == true){

While(count < 10)

Print(count)

}

Then both the If, and While objects will contain the above loop for triggering the parse. Basically, parsing a block will trigger the parsing of its contents, and blocks within it. This means that with just these two for loops, every command in any size of GNG script can be parsed.

### Code Generation

Once the syntax analysis has established that the code is correct, it can go on to create the actual assembly instructions to be executed. It will systematically go over each object, and generate the required assembly instructions, and simply output them to a file in order.

for (Struc structure: code) {

    for (String instruction: structure.buildAndGetInstructions()) {

        if (!append(instruction)) {

            Error.customError("Could not write to file!");

            return false;

        }

    }

}

The first loop here will iterate over each object, and the second one will build and get the assembly instructions created which will return the assembly instructions in the form of an ArrayList, and so outputs each one to the file as it comes.

## Programming Construct Implementations

### Variables

### Branching

### Looping

## Garbage Control

As this is an 8-Bit system, it only has 256 addressable memory locations, and no physical stack hardware to manage, and as such, there is only basic garbage control required. In addition, only integer values are accepted, and so datatype managing is also not needed. The main processing that occurs here is assigning memory addresses to created variables, and providing a memory location if an existing variable name is provided.

While very basic, is still essential to remove the chance of data collisions or memory addresses being overwritten unexpectedly which will cause huge issues with program execution. The code for the three main features of the garbage control are as follows:

public static int has(String varName){

if(variables.containsKey(varName))

        return variables.get(varName);

    return -1;

}

public static boolean newVariable(String name){

    if(variables.containsKey(name) || full)

        return false;

    variables.put(name, memoryHolder);

    memoryHolder++;

    full = memoryHolder >= 255;

    return true;

}

public static Integer getAddress(String varName){

    return variables.get(varName);

}

The 3 main functions here are “has”, “newVariable”, and “getAddress”.

The “has” function simply returns the address of a given variable name, or -1 if the given variable does not exist. The “newVariable” function is used when defining a new variable, and will add one if there is space (less than 255) available address. This is because address 255 (11111111) is reserved for multiple term calculations. So there are 255 (including 0) addressable memory locations available for general use.

“getAddress” is a simplified version of the “has” function which is shorter as it is only used after validation of the name has occurred. It is a last line of defence against syntax errors causing an error injection into the actual assembly output. If the compiler gets to this point, and there is still somehow an invalid variable name, a -1 would be displayed into the output assembly file. Instead of allowing this to happen, an error will be triggered.

## Error Handling

## Operating System

The operating system consists of a huge list of raw binaries across 3 EEPROM (Electrically Erasable Programmable Read Only Memory) chips. The reason for needing 3, is due to the vast amount of modules present on the computer. There are 24 primary functions of the computer, and each EEPROM chip has 8 outputs, so each can control 8 functions of modules. These controls are incredibly basic, and cannot do much of anything on their own; the combination of instructions and logic is what makes computers so powerful.

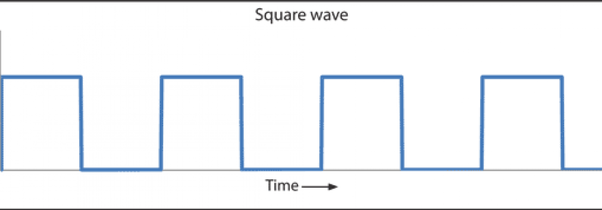
An example of a module is the ‘A’ Register, this is the primary register used for arithmetic, comparisons, and is the default location for loading data. The computer does no know this, and it is the binary in the OS which makes this the case. The operations that the ‘A’ Register can perform are simple loading a value from the data bus, and outputting a value onto the data bus. As such, the ‘A’ Register has 2 controls, and there are 3 registers on in the computer, taking up the first 6 controls. It is the granularity of each possible operation undertaken by every module that creates the need for so many EEPROM chips to be used.

The computer itself does not recognise the number 1 or the number 0, they are simply symbols, much in the same way symbols are used in everyday life to depict something else, like π meaning 3.14. Electronics use voltage differences as signals, and these difference are defined as 1 and 0, the computer uses chips which all use +5v and 0v as the 1 and 0 signals. These charges are the exact same as someone opening and closing a switch to turn on and off lightbulb, the only two differences are that the computer’s control logic is the one flipping these switches, and the signals can be used as controls as well as to power something.

Calendar

Description automatically generatedThe OS has absolute control over all of the modules, to so much of an extent that the OS (if programmed incorrectly) could cause damage to the data integrity, or even physical damage to the silicon on the chips. This could be done by overloading the data bus, which could lead to short-circuits burning out chips, and data being saved/loaded incorrectly.

The above instruction is used to load a literal value into the ‘A’ Register. The Instruction takes advantage of the fact that the EEPROM chips used have 13 address lines, 8 of these are used for the instruction itself, meaning that the computer can have a maximum of 256 commands. The remaining 5 of the 13 available bits are used for “Micro Steps” which are the commands to run within each instruction. Each of these micro steps take exactly one clock cycle to run. They are run on the falling edge of the cycle, and the setup time is from the rising edge up until the falling edge starts.



Set up

Execute

## The Hardware

As this project is such a low level, there would be an injustice in not including some of the hardware in this report. While a lot of the hardware is complex and not relevant, there is also a great deal of digital logic which is linked to the software goings on.

A screenshot of a computer

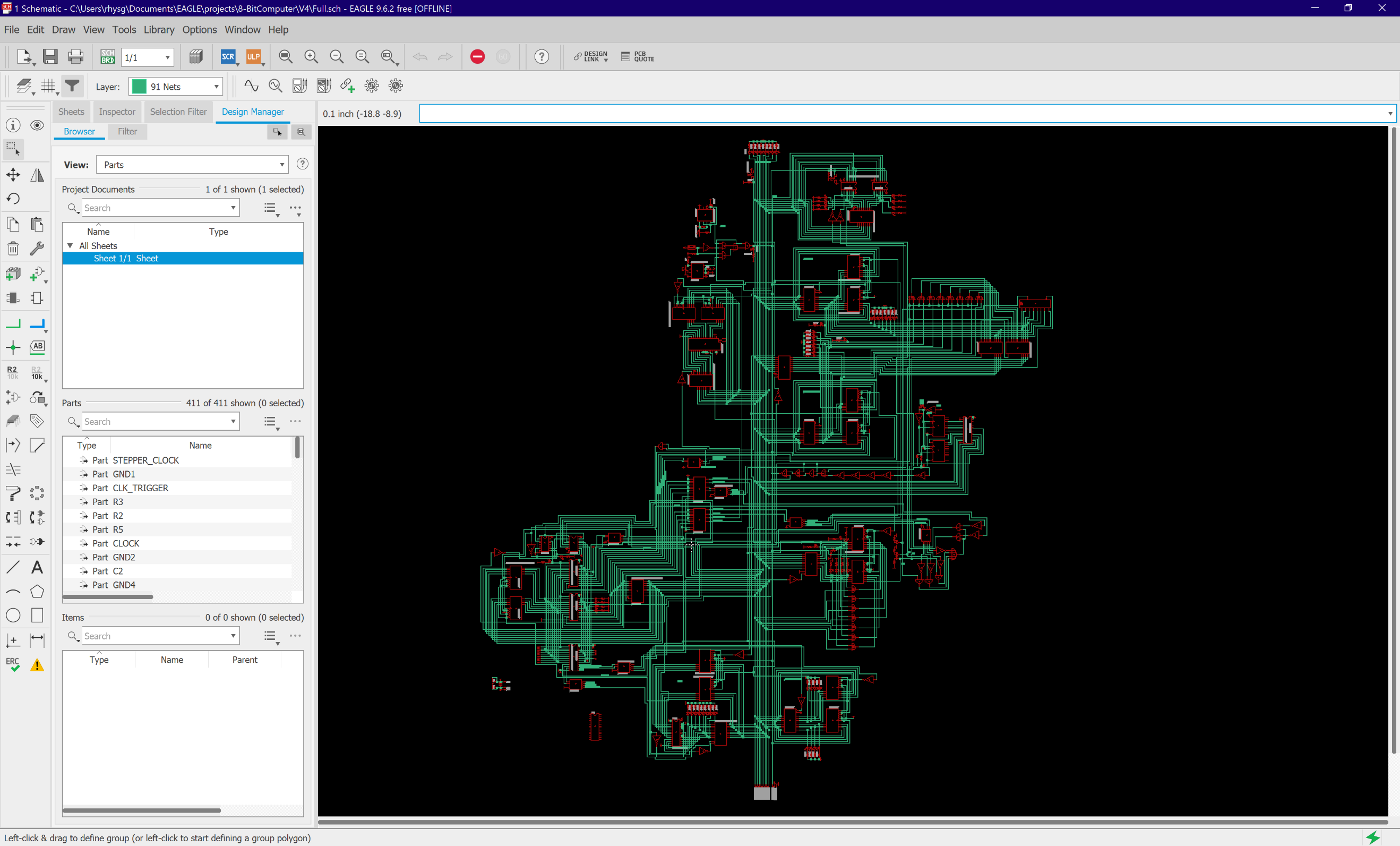
Description automatically generatedOne such example of good digital logic is the logic which controls the branching in the computer.

Diagram

Description automatically generatedThe central square is a “74ls139” chip which turns 2 input into 4 outputs. This is achievable as 11 in binary is 3, and including 0 makes 4 outputs:

|  |  |
| --- | --- |
| A | B |
| 0 | 0 |
| 0 | 1 |
| 1 | 0 |
| 1 | 1 |

From the datasheet of this chip, we can see that the 4 outputs labelled o0 – o3 go low (L) when active. This can be used as signals throughout the computer. In the branching logic, the two inputs are “Loop Ctrl A” and “Loop Ctrl B”, and the four possible branches are an unconditional branch, branch when the ‘A’ Register = 0, branch on a carry (result of calculation is greater than 255), and branch when the ‘A’ Register is greater than 0. The signal selector then passes through an AND gate along with the associated flag. Take BRZ (Branch when ‘A’ Register is 0) for example. BRZ is linked to output 2, so Loop Ctrl A will be High (H or 1), and Loop Ctrl B will he Low (L or 0). After the selector, this will tie O1 low, as this is the line being selected. This then goes through the not gate to convert the 0 to a 1, and into an AND gate, the other line on the AND is attached to the Zero Flag. This flag is high when the value of the ‘A’ Register is 0. If at this point the ‘A’ Register is 0, then the AND gate would return high, as both the zero flag and BRZ selector are high. It does not matter about the state of the other relevant flags, because all of the other lines coming out of the selector are high, and as such, they go through the not gate, resulting in one of the lines in each of the AND gates being low. Due to the nature of AND gates, with this 0 going in, they will all result in 0 apart from the one which is selected. This then enters a 4 input OR gate, and then off to the program counter to loop to a given step.



1

2

4

5

6

7

9

8

10

11

14

13

12

3

This is the circuit diagram for the whole computer. It looks like a full city of road and buildings, and thinking like this is helpful to get your bearings of the diagram.

|  |  |  |
| --- | --- | --- |
| Number | Part | Explanation |
| 1 | Program Counter (PC) | Stores the current step to be used as the memory address, and incremented when instructions are executed |
| 2 | Clock (CLK) | To keep timings and trigger certain events |
| 3 | ‘A’ Register | A general purpose register which all of the branching is based upon |
| 4 | Arithmetic and Logic Unit (ALU) | Can add and subtract integers based on the ‘A’ and ‘Calc’ Registers |
| 5 | Memory | Stores the program memory |
| 6 | ‘Calc’ Register | A secondary register to work on calculations, and can be used for temporary storage |
| 7 | PS/2 Keyboard Interface | Takes inputs from a PS/2 keyboard and can output it to the database for use throughout. |
| 8 | Instruction logic Data bus Interface | Enables the Instruction logic to interface with the data bus |
| 9 | Micro Step Counter | A secondary program counter to keep track of the steps within each instruction |
| 10 | Branching logic | To control the branches and to trigger them with the correct signals and conditions |
| 11 | Instructional Logic and OS | Contain the control signal lines, and Operating System Binaries |
| 12 | Toggleable Flag Register | Stores the built in and general-purpose flags which are toggleable |
| 13 | RAM | 256 addressable bytes with 8 bit address, and 8 bit data direct to the data buse |
| 14 | Parameter Register | A general-purpose register to store the parameter provided by each instruction. |

## Points of Interest

### Functionality vs. Performance

It is a general fact that high-level languages are less performant than low-level languages [reference], but it is rare that a good reason with examples is provided, other than to state that there are more layers involved, and high level is further away from the computer hardware. This project demonstrates with good examples how these inefficiencies occur and why they still exist even in today’s computer regardless of the fact that we are aware of the problems.

One such example of how these inefficiencies are using in high level languages is in my implementation of calculations during conditions and when setting values to variables. Take the following line:

If ( 5 + 6 == 3 ) // do something

The computer will need to complete the sum before being able to establish if the condition is true. One would think that the following assembly instructions would be correct for the line:

&LDA 5

&ADD 6

&SUB 3

BRZ 5

BRA 6

; do something

However, this is not the case, as the language must cater for any eventuality of program, and must keep the program running correctly, and have memory managed properly. The eventuality that must be accounted for is if there are calculations on the right hand side of the “==” for example:

If ( 5 + 6 == 2 + 8 ) // do something

After the first calculation is complete, the ‘A’ Register now contains the 11 which is need to be tested against. By now undertaking the second calculation (2 + 8), this 11 will be lost if it is not saved. Furthermore, if it is done this way around, the program will be even less performant. Lets go through what would need to happen:

* Load 5, and add 6
* Store this 11 in RAM
* Load 2 and add 8
  + As loading a value puts it in the A register, a straight load cannot be done here, as then the 10 from the previous calculation would be lost.
  + Also, a subtract cannot be done now either, as then it would be 11 – 10, it doesn’t matter much in this situation, as all the program is looking for is if the difference is 0, but in scenarios including “<” and “>”, the order is very important.
* Store this 10 in RAM
* Load the 11 from before
* Subtract the 10 from before
* Make the checks

We can see here this is a long process, and can be made slightly better, as if done in reverse, the second storing process is not required:

* Load 2, and add 8
* Store this 10 in RAM
* Load 5 and add 6
* Subtract the 10 from earlier
* Make the checks

While longer than the original statement, it is a lot shorter. The actual assembly instructions that are generated from the compiler are as follows:

&LDA 3

STO 255

&LDA 5

&ADD 6

SUB 255

BRZ 7

BRA 7

The condition length is the first 5 lines here, and compared to the first one with the bespoke logic of length 3, we can see that there has been a reduction in performance on purpose. If the code was written for the bespoke problem, then this performance issue could be mitigated, but with the universal and generalised nature of high level languages, this type of issue will be rife.

### Parsing Recursion

### Polymorphic Behaviours

Following the above example of the &LDA command running, in the OS EEPROM chips, at address: “0000000000010” is the value “000000000000000000010001”. It is obvious at this point why programming languages are required, because that binary is completely impossible to debug if ever there was an issue with the code. However, within the table, we can see exactly what is happening. Where the 1’s are in the instruction, that is where the controls are being run, the fields the 1’s are referring to on the first instruction of the &LDA command are: PCO, and MAI. These are abbreviations created for the commands they control. PCO is Program Counter Out, and MAI is Memory Address In. This means the that Program Counter will output the current step the computer is on and the Memory module will take that value in. This is all that is done for that step, and at the next clock tick, the micro address counter will increment by one, and the new address being looked at inside the EEPROM chips is now “0000100000010”. This leads on to another set of binary, and new commands for the modules to listen to.

The OS is built using the method of “Fetch, Decode, Execute” [Reference] which means that the command is fetched from memory, decoded using the EEPROM chips, and then executed by the signal logic to tell the relevant modules what to do. This is done in the first 4 steps of each command, which is followed by the unique code to the instruction, hence doing something different per instruction.

## Summary

Write a short summary at the end of each chapter. Do not use the words ‘In summary’, we know what it is from the title.

# Testing

## Introduction

Each of your chapters should have an introduction to tell your readers what they will find in the chapter. In this chapter you should introduce your test strategy – how have you tested your artefact. You should also talk about user testing. How did you test with real people? How did you select them? What did you ask them to do? What ethical considerations did you adhere to? In this chapter you will also discuss how you have carried out an evaluation of your artefact. This is not the same thing as a total project evaluation.

## Functional Testing

To add a caption to a table, either select the whole table (e.g. by clicking on the + symbol in the upper left corner of the table), right-click it and choose ‘Insert Caption’ or click in any table cell and select ‘References’ -> ‘Insert Caption’ from the ribbon menu. Choose ‘Table’ as label and ‘above the item’ as position. Add the caption text in the box, separated with a dash as the example below shows.

Table - Test Results

|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

## Non-Functional Testing

Content goes here.

## User Testing

Content goes here.

## Evaluation

Content goes here.

## Summary

Write a short summary at the end of each chapter. Do not use the words ‘In summary’, we know what it is from the title.

AND USER TESTING

# Evaluation, Conclusions and Future Work

## Project Objectives

Summarise what you have achieved.

## Self-Evaluation

This section is about yourself. Be honest. Look at where you were situated at the beginning of the project and where you are now. What have you learnt on a personal level, what have you found out about yourself? Try to reflect upon individual goals, experiences, and incidents. No one is perfect, and it is very likely that you will recall both good and bad experiences.

The purpose of the evaluation process is to highlight strengths, correct performance weaknesses, and develop unused skills and abilities. To do this, you must be willing to recognise areas that need improvement or development.

## Project Evaluation

Stand back and evaluate what you have achieved and how well you have met the objectives. Evaluate your achievements against your objectives in section 3.2. Demonstrate that you have tackled the project in a professional manner.

(The previous paragraph demonstrates the use of automatic cross-references: The ‘3.2’ is a Cross-reference to the text in a numbered item of the document, it is not literal text but a field. The number that appears here will change automatically if the number on the referred-to section is altered, for example if a chapter or section is added or deleted before it. Cross-references are entered using Word's Insert or References menu. Cross-references are set to update automatically when printed but may not do so on-screen beforehand; you can update a field manually on-screen by right-clicking on it and selecting Update field from the pop-up menu or by selecting the whole document and pressing F9.)

## Applicability of Findings to the Commercial World

Summarise what you have achieved and how it can apply to the commercial world.

## Conclusions

Summarise what you have achieved. Do not use the words ‘In conclusion’ or ‘to conclude’ or any derivative of those. We know this is the conclusions from the title.

## Future Work

Explain any limitations in your results and how things might be improved. Discuss how your work might be developed further. Reflect on your results in isolation and in relation to what others have achieved in the same field. This self-analysis is particularly important. You should give a critical evaluation of what went well, and what might be improved.

Graphics card, Stack trace debugging tool, keyboard interface

# References

Cuevas, R., Cuevas, Á., Cabellos-Aparicio, A., Jakab, L., & Guerrero, C. (2010a). A collaborative P2P scheme for NAT Traversal server discovery based on topological information. *Computer Networks, 54*(12), 120-122.

Cuevas, R., Cuevas, Á., Cabellos-Aparicio, A., Jakab, L., & Guerrero, C. (2010b). Book Section. In J. D. Kramer (Ed.), *This is a another Book released in the same year* (Vol. 54, pp. 150-180). London: PressAwesome.

Greene, D., & Williams, P. C. (1997). *Linear Accelerators for Radiation Therapy* (2nd ed.). Bristol and Philadelphia: IOP Publishing Ltd.

Jacobson, J., & Andersen, O. (1997). Software Controlled Medical Devices. In *SP Report 1997:11.* Sweden: Swedish National Testing and Research Institute.

Ji, H., & Turner, K. J. (1999). Specification and Verification of Synchronous Hardware using LOTOS. In J. Wu, S. T. Chanson, & Q. Gao (Eds.), *Formal Methods for Protocol Engineering and Distributed Systems (FORTE XII/PSTV XIX)* (pp. 295-312). London: Kluwer Academic Publishers.

Turner, K. J., & Jennings, M. A. (2002). The Rules of Sailing Races for Hand-Held Devices. *J. Navigation, 23*(5), 114-240.

University of Stirling. (2002). *Computing Science and Mathematics Research*. Retrieved March 01, 2017, from http://www.cs.stir.ac.uk/research

# Appendix 1 – Project Proposal

Your first appendix should be a copy of your Project Proposal.

You may have one or more appendices containing detail, bulky or reference material that is relevant though supplementary to the main text: perhaps additional specifications, tables or diagrams that would distract the reader if placed in the main part of the dissertation. Make sure that you place appropriate cross-references in the main text to direct the reader to the relevant appendices.

Do not blindly include all your code in the appendix or the body. Only include the parts you refer to in the report. You can put those parts either in the appendix or in the body (e.g. in the “Implementation” part).

# Appendix 2 – Technical Plan

Your second appendix should be a copy of your Technical Plan.

You may have one or more appendices containing detail, bulky or reference material that is relevant though supplementary to the main text: perhaps additional specifications, tables or diagrams that would distract the reader if placed in the main part of the dissertation. Make sure that you place appropriate cross-references in the main text to direct the reader to the relevant appendices.

Do not blindly include all your code in the appendix or the body. Only include the parts you refer to in the report. You can put those parts either in the appendix or in the body (e.g. in the “Implementation” part).

# Appendix 3 – Title of Appendix

You may have one or more appendices containing detail, bulky or reference material that is relevant though supplementary to the main text: perhaps additional specifications, tables or diagrams that would distract the reader if placed in the main part of the report. Make sure that you place appropriate cross-references in the main text to direct the reader to the relevant appendices.

Do not blindly include all your code in the appendix or the body. Only include the parts you refer to in the report. You can put those parts either in the appendix or in the body (e.g. in the “Implementation” part).