CHAPTER ONE

1.1 INTRODUCTION

1.2 BACKGROUND OF THE STUDY

Assembly language is a low-level language for a computer, or other programmable device specific to particular computer architecture in contrast to most high level programming languages which are generally portable across multiple systems (William 1962). Assembly language is converted to executable machine code by utility program referred to as an assembler like NASM, MASM, and Easy68k.

Virtual Machine (VM) is an emulation of a computer system. Virtual machines are based on computer architectures and provide the functionality of a physical computer. Their implementation may involve specialized hardware, software or a combination (Ravi 2005). Virtual machines resemble real processors, but are implemented in software. They accept as input a program composed of a sequence of instruction called bytecode. Virtual machines has almost all the features of a real computer such as registers, stacks, memory address, interrupts, and graphical user interface (GUI). The desire to run multiple operating systems was the initial motive for virtual machines, so as to allow time-sharing among several single-tasking operating systems (Smith 2016). Examples of virtual machine platform include Android Dalvik Virtual Machine, VMware Workstation, VMware Fusion, Windows Virtual PC, Oracle VM Server for Spark, and Virtual Box. Major uses of virtual machine includes running Linux applications on windows, running android applications on PC(Personal Computers), and running PlayStation games on windows.

Waves assembly language is a new assembly language that has never existed before and some of its syntax is borrowed from Arm assembly language, Java Assembly language, and Spark assembly language. It is implemented to compile and run in a web browser. Waves assembly language platform is made up of an assembler, editor, and virtual machine. Waves virtual machine is a register based virtual machine in which almost all operations are performed on registers instead of stacks. Waves virtual machine registers are actually internally mapped to stacks. The reason for using register based architecture is that register operations are easier to understand as it is straight forward. The Waves assembler turns the assembly language code into machine level language known as bytecode. The executable code generated by the assembler will then be fed to the virtual machine for execution.

1.2 PROBLEM STATEMENT

Due to use of desktop computer for native compilation of assembly languages, users usually encounter the following problems:

- Native assembly languages platform are not user friendly and are prone to error
- Existing assembly languages syntax is hard to learn
- Native assembly languages are not designed for beginner programmers
- Native assembly languages are not portable across multiple systems.

Waves assembly language will solve this problem by providing a user friendly environment for users to learn. Waves assembly language is also portable across multiple devices that has a web browser.

1.3 AIM AND OBJECTIVES OF THE STUDY

The aim of this project is to design and implement Waves Assembly Language for Web Platform and the objectives are:

- To design and implement a new assembly language named Waves.
- To create an English like assembly language that is easy and fun for beginners to learn.
- To make Waves assembly language compile and run in a web browser.
- To create an assembler and virtual machine for Waves assembly language.

1.4 SIGNIFICANCE OF STUDY

Waves assembly language is simple and useful tool for learning the basics of assembly language programming. It provides a starting point for beginner system programmers wanting to go into assembly language programming. It has English like syntax that makes it easy and fun for beginner programmers. Waves Integrated Development Environment (IDE) provides a controlled environment for coding that is forgiving such that when users make mistakes, users can easily correct their mistakes without breaking something on the system.

Students who are willing to learn assembly languages may find it difficult probably because of unavailability of a Personal Computer (PC) or because of the difficult syntax of most assembly languages. Students can easily make use of their mobile phones to access Waves Integrated Development Environment using their web browser. Students can also easily learn Waves assembly language in order to have an idea of what assembly language programming is all about and that will help the students to be able to pick up other assembly languages easily.

1.5 SCOPE OF STUDY

The scope of this project is to design and implement Waves assembly language that compiles and run on any device that has a web browser.

1.7 LIMITATION OF THE STUDY

The limitation of this project is that it can only compile and run in a web browser and it does not support GUI (Graphical User Interface) programming.

1.8 DEFINATION OF TERMS

- **HTML**

HTML stands for Hyper Text Markup Language. It is the set of markup symbols or codes inserted in a file intended for display on a webpage. The markup tells the browser how to display the web pages words and images for the user.

- **CSS**

CSS stands for Cascading Style Sheet. It is the language used to describe the presentation of a web pages, including colors, layout, and fonts. It allows one to adapt the presentation to different types of devices, such as large screens, small screens, or printers.

- **ASSEMBLY LANGUAGE**

Assembly language is a programming language that consists of instructions that are mnemonic codes for corresponding machine language instruction. It is a language for a particular computer architecture which is used for creating device drivers and processor specific applications.

- **IDE**

IDE stands for Integrated Development Environment. It is a software application that provides facilities to computer programmers for creating, editing, and debugging software applications.

- **JAVASCRIPT**

JavaScript is an object oriented computer programming language commonly used to create interactive effects within web browsers.

- **ASSEMBLER**

An assembler is a type of computer program that interprets software programs written in assembly language into machine language, code and instructions that can be executed by a computer.

- **NATIVE LANGUAGE**

It is a language that known as host language. It is the language understood by the computer without being converted into another form. It is a code written to run on a specific processor.

- **EMULATION**

Emulation is the use of an application or program or device to imitate the exact behavior of another program or device.

- **SIMULATION**

This is used to provide realistic imitation of a certain object without accurately reproducing all of its features.

- **ARCHITECTURE**

This is a formal specification of an interface in the system, including the logical behavior of resources managed via the interface.

- **REGISTER**

It is a computer memory that is used to quickly accept, store, and transfer data and instructions that are being used immediately by the CPU.

CHAPTER TWO

2.0 LITERATURE REVIEW

This chapter reviews the literatures that are related to this project work. It embraces all the different fields that are related to the design and implementation of Waves assembly language for web platform.

2.1 PROGRAMMING LANGUAGES

A programming language is a formal language that specifies a set of instructions that can be used to produce various kinds of output. Programming languages generally consist of instructions for a computer. Programming languages can be used to create programs that implement specific algorithms. (Koetsier 2001).

The earliest known programmable machine preceded the invention of the digital computer and is the automatic flute player described in the 9th century by the brothers Musa in Baghdad. From the early 1800s, programs were used to direct the behavior of machines such as Jacquard looms and player pianos. Thousands of different programming languages have been created, mainly in the computer field, and many more still are being created every year. Many programming languages require computation to be specified in an imperative form (as a sequence of operations to perform) while other languages use other forms of program specification such as the declarative form (the desired result is specified, not how to achieve it). The description of a programming language is usually split into the two components of syntax (form) and semantics (meaning). (Rojas 2000).

Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

2.1.2 CHARACTERISTICS OF A PROGRAMMING LANGAUGE

Programming languages differ from natural languages in that natural languages are only used for interaction between people, while programming languages also allow humans to communicate instructions to machines.

2.1.2.1 ABSTRACTIONS

Programming languages usually contain abstractions for defining and manipulating data structures or controlling the flow of execution. The practical necessity that a programming language support adequate abstractions is expressed by the abstraction principle; this principle is sometimes formulated as a recommendation to the programmer to make proper use of such abstractions.

2.1.2.2 EXPRESSIVE POWER

Languages can classified by the amount of computations that can be expressed in that particular language. All Turing complete languages can implement the same set of algorithms. Markup languages like XML, HTML, which define structured data, are not usually considered programming languages. Programming languages may, however, share the syntax with markup languages if a computational semantics is defined.

The term computer language is sometimes used interchangeably with programming languages. However, the usage of both terms varies among authors, including the exact scope of each. One usage describes programming languages as a subset of computer languages. For instance, markup languages are sometimes referred to as computer languages to emphasize that they are not meant to be used for programming.

Formal specification languages are just as much programming languages as are the languages intended for execution. Another usage regards programming languages as theoretical constructs for programming abstract machines, and computer languages as the subset thereof that runs on physical computers, which have finite hardware resources. (John C. Reynolds 2008).

2.1.3 TYPES OF PROGRAMMING LANGUAGES

Programmers write instructions in various programming languages, some directly understandable by computers and others requiring intermediate translation steps. Hundreds of computer languages are in use today. These can be divided into three general types:

2.1.3.1 ***MACHINE LANGUAGE***

Machine language is the natural language of a computer and such is defined by its hardware design. Machine languages generally consist of strings of numbers (ultimately reduced to 1s and 0s) that instruct computers to perform their most elementary operations one at a time. Machine languages are machine dependent (i.e a particular machine language can be used on only one type of computer).

2.1.3.2 LOW LEVEL LANGUAGE

Machine Language were simply too slow and tedious for most programmers, instead of using strings of numbers that computers could directly understand, programmers began using English like abbreviations to represent elementary operations. These abbreviations form the basis of Low Level Language. In low level language, instructions are coded using mnemonics. E.g. DIV, ADD, SUB, MOV. Assembly language is an example of a low level language.

An assembly language is a low-level language for programming computers. It implements a symbolic representation of the numeric machine codes and other constants needed to program a particular CPU architecture. This representation is usually defined by the hardware manufacturer, and is based on abbreviations (called

mnemonics) that help the programmer to remember individual instructions, registers, and cache. An assembly language is thus specific to a certain physical or virtual computer architecture (as opposed to most high-level languages, which are usually portable).

2.1.3.2 HIGH LEVEL LANGUAGE

Computers usage increased rapidly with the advent of assembly languages, but programmers still have to use many instructions to accomplish even the simplest tasks. To speed up the programming process, high level language were developed in such a way that simple statements could be written to accomplish substantial tasks which can be translated using translator programs called compilers which convert high level language programs into machine language. High level language allows programmers to write instructions that look almost like every day English and contain commonly used mathematical notations.

2.1.3 HISTORY OF PROGRAMMING LANGUAGES

2.1.3.1 EARLY DEVELOPMENTS

The earliest computers were often programmed without the help of a programming language, by writing programs in absolute machine language. The programs, in decimal or binary form, were read in from punched cards or magnetic tape or toggled in on switches on the front panel of the computer. Absolute machine languages were later termed first-generation programming languages (1GL) .

The next step was development of second-generation programming languages (2GL) or assembly languages, which were still closely tied to the instruction set architecture of the specific computer. These served to make the program much more human-readable and relieved the programmer of tedious and error-prone address calculations.

The first high-level programming languages, or third-generation programming languages (3GL), were written in the 1950s. An early high-level programming language to be designed for a computer was Plankalkül, developed for the German Z3 by Konrad Zuse between 1943 and 1945. However, it was not implemented until 1998 and 2000.

2.1.3.2 ***REFINEMENT***

The increased use of high-level languages introduced a requirement for low-level programming languages or system programming languages. These languages, to varying degrees, provide facilities between assembly languages and high-level languages and can be used to perform tasks which require direct access to hardware facilities but still provide higher-level control structures and error-checking.

Moreover, the period from the 1960s to the late 1970s brought the development of the major language paradigms now in use.

*2.1.3.3 **CONSOLIDATION AND GROWTH***

In 1980s, consolidation of many programming languages occurred which led to the combination of Object oriented programming and system programming into C++. Ada programming language was also standardized by the United States. Ada is a system programming language that was derived from Pascal programming language and is mostly used by the defense contractor.

Another important trend in 1980s was an increased focus on large-scale systems through the use of large-scale organizational units of code. Programming languages evolution also continued, in both military and research.

2.1.4 SYSTEMS PROGRAMS

System program can be known as a program, an operating system, compiler, or utility program that controls some aspect of the operation of a computer. System programs are not usually part of the overall operation system.

2.1.4.1 FILE MANAGEMENT PROGRAMS

File management program is a type of system program that is used to create, delete, copy, rename, print, dump, list, and generally manipulate files and directories.

2.1.4.2 STATUS INFORMATION PROGRAMS

Status information program is a type of system program that is simply request the date and time, and other simple requests.

2.1.4.3 FILE MODIFICATION PROGRAMS

File modification programs are programs such as text editor that are used to create, and modify files.

2.1.4.4 COMMUNICATION PROGRAMS

Communication programs provide the mechanism for creating a virtual connect among processes, users, and other computers. Example of communication includes email and web browser.

2.2 VIRTUAL MACHINE

A virtual machine (VM) is an emulation of a computer system, they are based on computer architectures and provide the functionality of a physical computer. Their implementation may involve specialized hardware, software or a combination.
Virtual machines resemble real processors, but are implemented in software. Virtual machines have their own machine language instructions called bytecode. Bytecode is an executable code for a virtual processor that is portable across multiple system that has the virtual machine.

Virtualization has become an important tool in computer system design, and virtual machines are used in a number of sub disciplines ranging from operating systems to programming languages to processor architectures. By freeing developers and users from traditional interface and resource constraints, virtual machines enhance software interoperability, system impregnability, and platform versatility.

2.2.1 **KIND OF VIRTUAL MACHINES**

2.2.1.1 SYSTEM VIRTUAL MACHINES (FULL VIRTUALIZATION VM)

System virtual machine provide a substitute for a real machine, they provide functionality needed to execute entire operating systems. A hypervisor uses native execution to share and manage hardware, allowing for multiple environments which are isolated from one another, yet exist on the same physical machine. Modern hypervisors use hardware-assisted virtualization, virtualization-specific hardware, primarily from the host CPUs.

2.2.1.2 PROCESS VIRTUAL MACHINES

Process virtual machines are designed to execute computer programs in a platform independent environment.

2.3 ASSEMBLY LANGUAGE

Assembly language is a low-level language for a computer, or other programmable device specific to particular computer architecture in contrast to most high level programming languages which are generally portable across multiple systems. Assembly language is converted to executable machine code by utility program referred to as an assembler like Nasm, Masm, and Easy68k.

However, each family of processors has its own set of instructions for handling various operations like getting input from keyboard, displaying information on screen and performing various other jobs. These set of instructions are called machine language instruction. Processor understands only machine language instructions which are strings of 1s and 0s. However machine language is too complex for using in software development. So the low level assembly language is designed for a specific family of processors that represents various instructions in symbolic code and a more understandable form.

2.4 ASSEMBLER

An assembler program creates object code by translating combinations of mnemonics and syntax for operations and addressing modes into their numerical equivalents. An assembler translates assembly language source code into machine instructions.

2.4.1 TYPES OF ASSEMBLER

2.4.1.1 Macro Assembler

A macro assembler includes a macro instruction facility so that (parameterized) assembly language text can be represented by a name, and that name can be used to insert the expanded text into other code.

2.4.1.2 Cross Assembler

A cross assembler is an assembler that is run on a computer or operating system (the host system) of a different type from the system on which the resulting code is to run (the target system). Cross-assembling facilitates the development of programs for systems that do not have the resources to support software development, such as an embedded system. In such a case, the resulting object code must be transferred to the target system, either via read-only memory (ROM, EPROM, etc.) or a data link using an exact bit-by-bit copy of the object code or a text-based representation of that code, such as Motorola S-record or Intel HEX.

2.4.1.3 High-level Assembler

A high-level assembler is a program that provides language abstractions more often associated with high-level languages, such as advanced control structures (IF/THEN/ELSE, DO CASE, etc.) and high-level abstract data types, including structures/records, unions, classes, and sets.

2.4.1.4 Micro assembler

A micro assembler is a program that helps prepare a micro program, called firmware, to control the low level operation of a computer.

2.4.1.5 Meta-assembler

A meta-assembler is a program that accepts the syntactic and semantic description of an assembly language, and generates an assembler for that language.

2.5 RELATED WORKS

2.5.1 DESIGN OF JAVASCRIPT ASSEMBLY X86 COMPILER AND EMULATOR FOR EDUCATIONAL PURPOSES (*Carlos Rafael Gimenes das Neves 2013*).

He designed JavaScript assembly X86 compiler and emulator for educational purpose and it runs in web browser. The design is mainly for the purpose of the class he taught Computer Architecture. The project site contains the implementation of his project in English language. He used HTML5 and CSS3 for the user interface and CodeMirror for the code editor interface.

The limitations of his project are that:

Not all instructions have been implemented

Only 9 registers available (debug and control registers are not available)

2.5.2 DESIGN OF JAVASCRIPT CHIP-8 EMULATOR

(*Alexander Dickson 2012*)

CHIP-8 is an interpreted programming language, and it runs in a virtual machine. It was made to allow video games to be more easily programmed for said computers. JavaScript chip-8 emulator is a virtual machine for chip-8 machine level language. The Chip-8 emulator uses a hexadecimal pad input and it uses a virtual keypad in order to be used on a desktop computer. HTML5 canvas is used for the graphical user interface, Web Audio API for the sound effect, and JavaScript for the virtual machine.

2.5.3 8-BIT ASSEMBLER SIMULATOR (*Marco Schweighauser 2015*)

8-bit assembler simulator is simulator written in JavaScript which provides a simplified assembler syntax (based on NASM) and it is used for simulating x86 like CPU. The emulator has an 8-bit CPU, 3 general purpose registers, 256 bytes of memory, a console output. The user interface is designed using HTML5 and CSS3, and the virtual machine and assembler are designed using JavaScript.

2.5.4 ARM ASSEMBLY LANGUAGE (*Howard Raggatt 1988*)

ARM assembly language is the easiest native assembly language in widespread use. ARM originally known as Acorn Risc Machine is a family of reduced instruction set computing architectures for computer processors, configured for various environment. ARM compiler is designed using C++ and C.

CHAPTER THREE

3.0 METHODOLOGY

3.1 OVERVIEW

This system was built using various programming languages such as JavaScript for building the compiler and virtual machine, HTML and CSS for the user interface, and Metro UI framework and Ace Edit framework for the windows like user interface.

The system allows users to run Waves assembly language using their existing devices and because of its high compatibility and simplicity, it requires only moderate technical know how to use the system. This system is compatible with all operating systems.

The phases of system development life cycle were adopted for the development of Waves assembly language for web platform. The phases involved include;

- Preliminary Study
- Feasibility Study
- System Analysis
- System Design
- Coding
- Testing
- Implementation
- Maintenance

3.1.1 System Investigation

This is the first phase of the software development cycle. In this stage, the process of investigations will be performed out on the existing assembly languages and how they are used.

In the existing assembly languages, user are expected to have a desktop computer and they are expected to have assembly language compiler installed on their systems.

3.1.2 Feasibility Study

This is the second stage in the system development life cycle. In this stage, the proposed system will be tested on factors based on Technical feasibility, Economical feasibility, and Operational/Behavioral feasibility.

3.1.2.1 **Operational cost**

The system working is quite easy to use and learn due to its simple and attractive interface. User requires no special training for operating the system.

3.1.2.2 **Economic cost**

During the investigation of the economic cost of the proposed system it was discovered that in terms of building, implementing, usage and maintenance of waves assembly language IDE(Integrated development environment), the system is cost effective as it can be used with existing devices and software.

3.1.2.3 **Technical cost**

The technical requirement for the system is economic as its requirements are flexible and compatible with most computers as most computers possess more than the required requirements.

Hardware required to use the developed system

- Any device
- RAM (512MB minimum)
- Processor (1GHz minimum)

Software required in order to use the developed system

- Any Operating System
- JavaScript enabled web browser

3.1.3 System Analysis

This stage involves 2 approaches:

3.1.3.1 **Problem of the Existing System**

In the existing system, desktop computer is being used to compile assembly languages and before any user can be able to develop any useful program with the existing assembly languages, the user must be very experienced. The existing system also assume that the users knows what they are doing such that when users make any mistake while writing a program in the existing assembly languages, it could make the whole desktop computer operating system crash.

3.1.3.2 **Proposed Alternative System**

To overcome the drawbacks of the existing system, the proposed system has been put in motion. This project aims to provide a portable and user friendly integrated development environment for users to interact with. Waves assembly language also has a simple and user friendly syntax with high level constructs.

3.1.4 System Design

This is the stage where the functionalities and requirement of the proposed system are mapped out and the steps to achieve it are stated out. The processes involved in designing the system are:

- Interface building: this deals with the designing of the user interface.
- Programming & application design: this deal with the design of the compiler and virtual machine.
- Compilation: it deals with the translation of the source code into an executable code.

3.1.4.1 **PROGRAM PSEUDOCODE AND PROGRAM FLOWCHART**

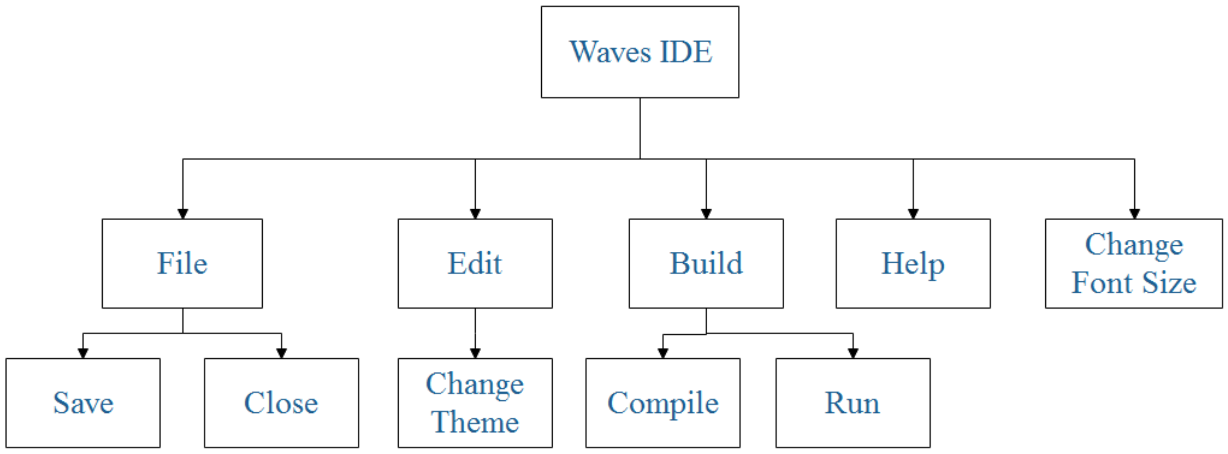


Figure 3.1 System Design for Waves Integrated Development environment

Figure 3.1 above shows the flow of the user interface functionality from the beginning to the end.

3.1.4.2 Pseudocode for the compiler

```
On Compile Button click:
Read the source code
    Identify what each word represents
    Convert each word to byte.
    Store the converted bytes as bytecode in a single variable
```

3.1.4.3 Pseudocode for the virtual machine

```
On Run Button click:
    Get the stored bytecode
    Decode each instruction from the bytecode
    Fetch the operands for each instruction
    Execute the instruction
    If program is interrupted then quit
```

3.1.5 System Coding

The system will be developed using HTML, CSS, and JavaScript programming language due to its compatibility with all browsers which is available on all Operating Systems. Sublime text was used as an Integrated Development Environment for coding of the user interface.

3.1.6 **System Testing**

The system was tested by some of the students of Computer Science Department of Federal College of Animal Health and Production Technology to know how easy and effective the system is.

Software is tested from different perspectives:

1. Code Testing: This involves step by step code analysis, testing and debugging. This process is done to prevent an error from occurring during program runtime. Automated testing is used for the code testing as manual testing is not sufficient to test all of its features.

2. Performance Testing: This involves the testing of the system and how it affects the performance of the computer system that is using it. Some factors considered are; memory usage and processing time.

3. System Testing: This involves the testing of the system with manual input of data. It is done to detect early malfunctions before final presentation of the system.

3.1.7 System Implementation

This is the stage whereby the IDE (Integrated Development Environment) is uploaded to the internet so that students and other users can have access to it anywhere.

CHAPTER FOUR

4.0 SYSTEM IMPLEMENTATION

4.1 DESCRIPTION OF THE NEW SYSTEM

The proposed Waves assembly language has an integrated development environment which consist of an editor, virtual machine, and assembler. In order to use Waves integrated development environment, the user only have to enter its web address in the web browser and the environment will be automatically loaded. The loaded environment contains an editor which users can directly input their programs. After the program has been inputted, the program will then be run by clicking the compile button and then the run button. A console window will then be displayed which contains the output of the inputted program.

The font-size and the theme of the environment can also be changed to suit the user. The user only has to click the button at the menu bar of the application.

4.2 Hardware and Software Used

Hardware Used

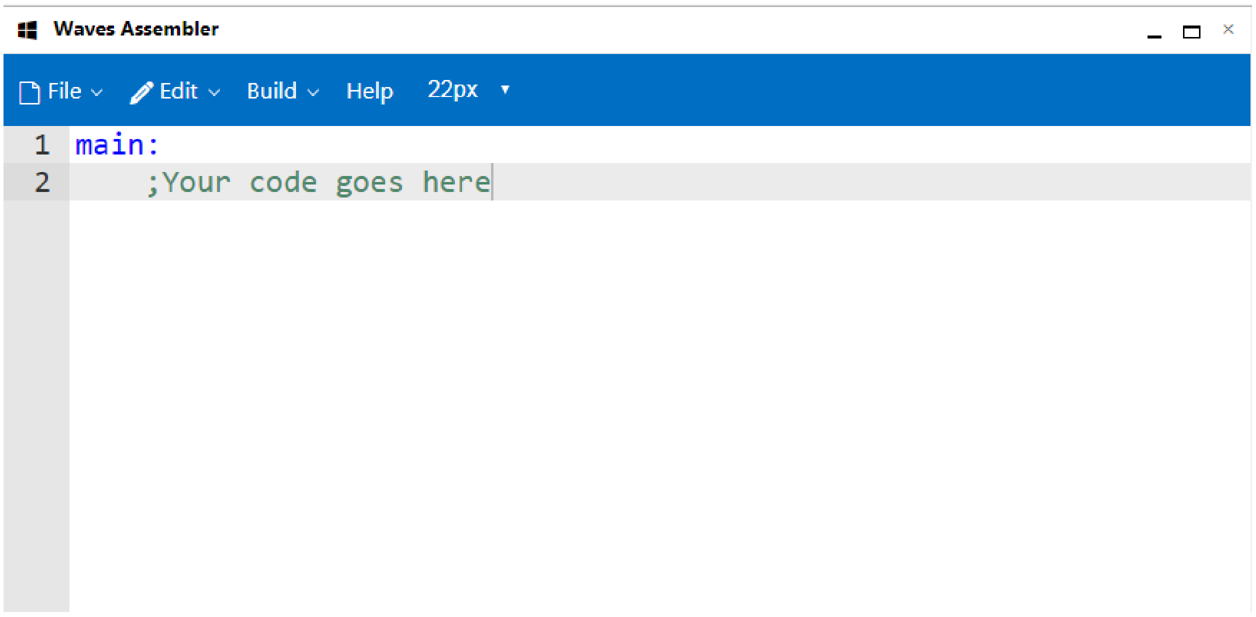
- HP Laptop NC800
- Random Access memory (RAM) 1.0GB
- 1.7GHZ processor

- 70GB hard drive

Software Used

- Windows 7 (Operating System)
- HTML5 (Hypertext Markup Language)
- JavaScript
- CSS3
- Xampp
- SlimJet Browser
- ACE Edit framework and Metro UI framework.

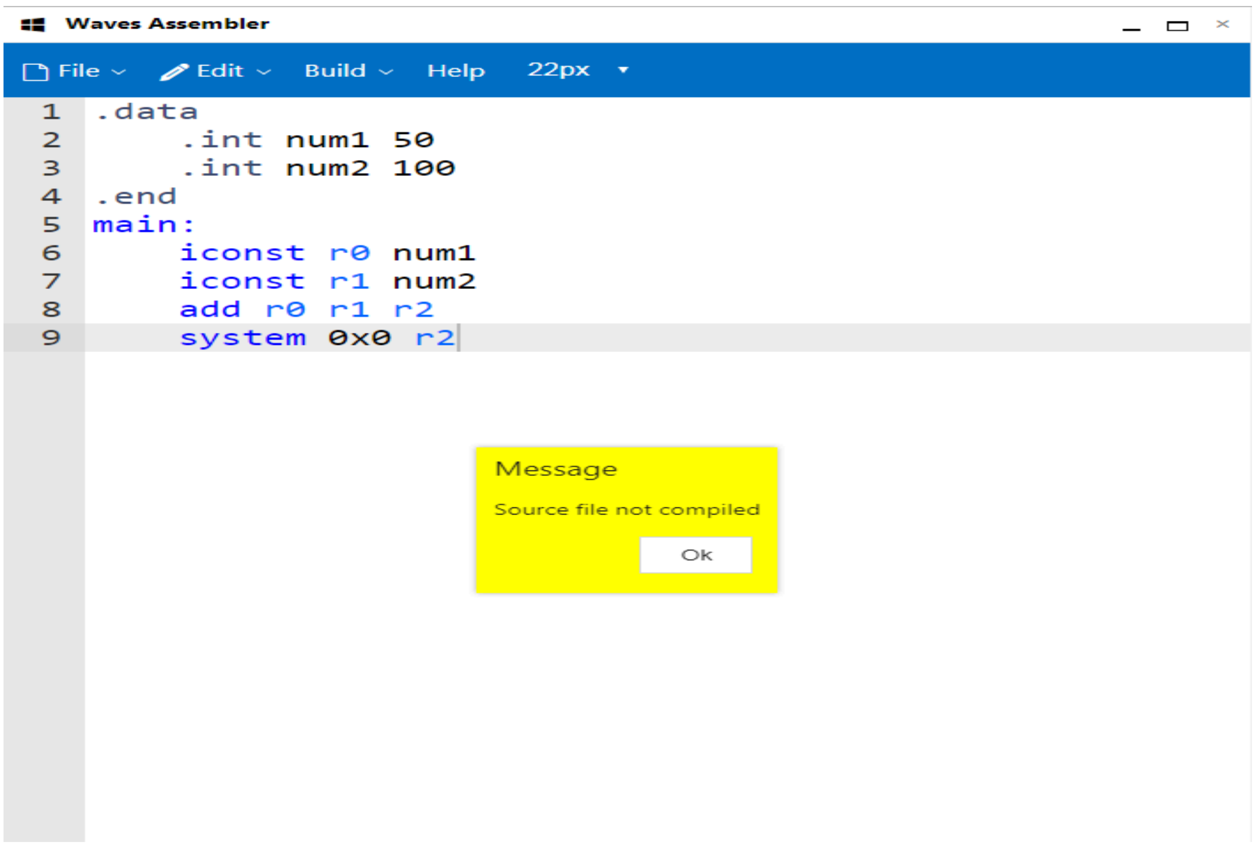
4.3 Input Design



```
main:
	;Your code goes here
```

Figure 4.1 Start-up Interface

Figure 4.1 above represents the start-up interface. It shows the start-up interface that contains the editor where the users will start inputting their code. Any code the users input must start below the main routine.



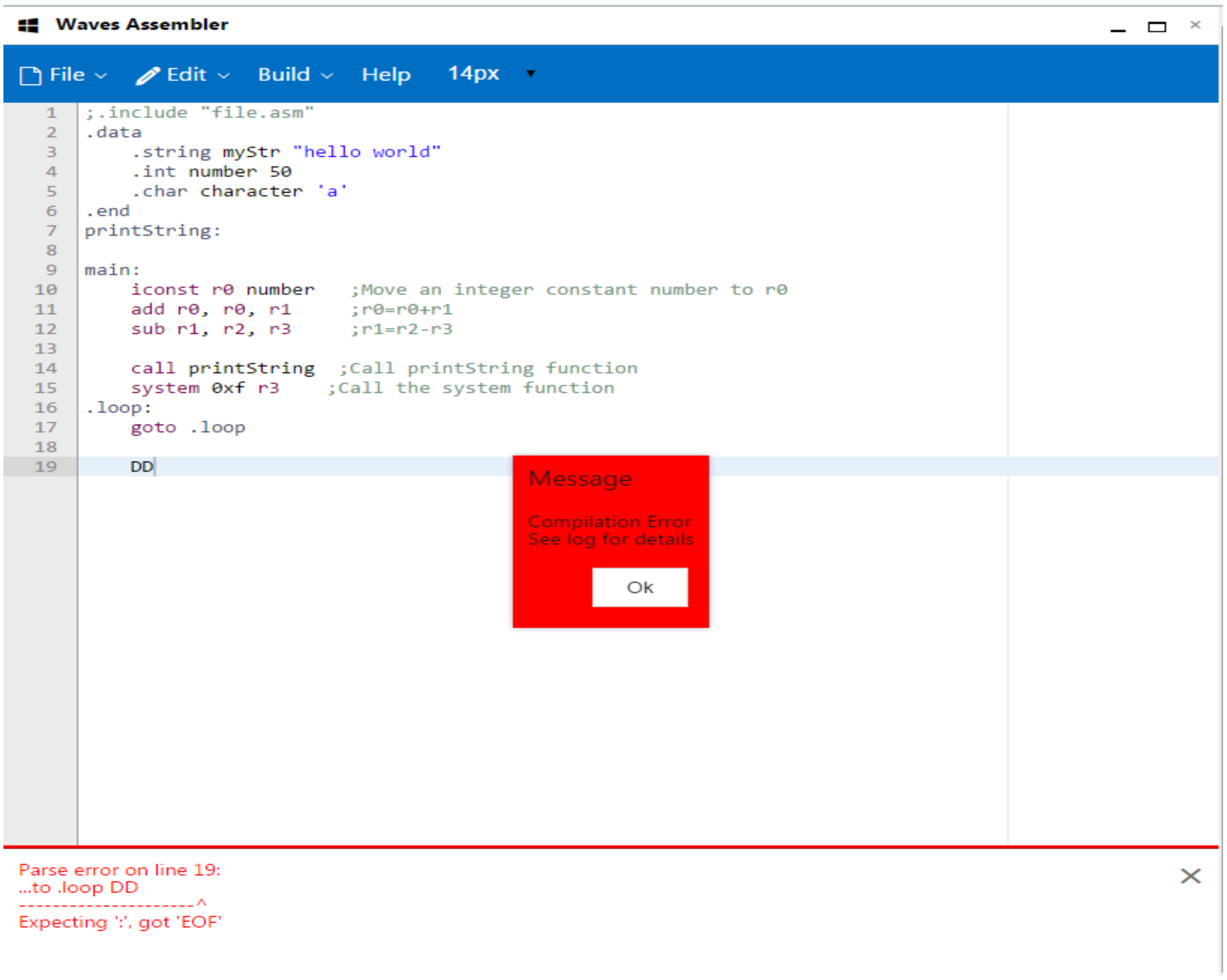
```
.data
      .int num1 50
      .int num2 100
.end
main:
      iconst r0 num1
      iconst r1 num2
      add r0 r1 r2
      system 0x0 r2
```

Message

Source file not compiled

Figure 4.2 Compilation Warning Popup window

Figure 4.2 above shows the compilation warning popup window. This popup window is shown when users try to run a program without compiling.



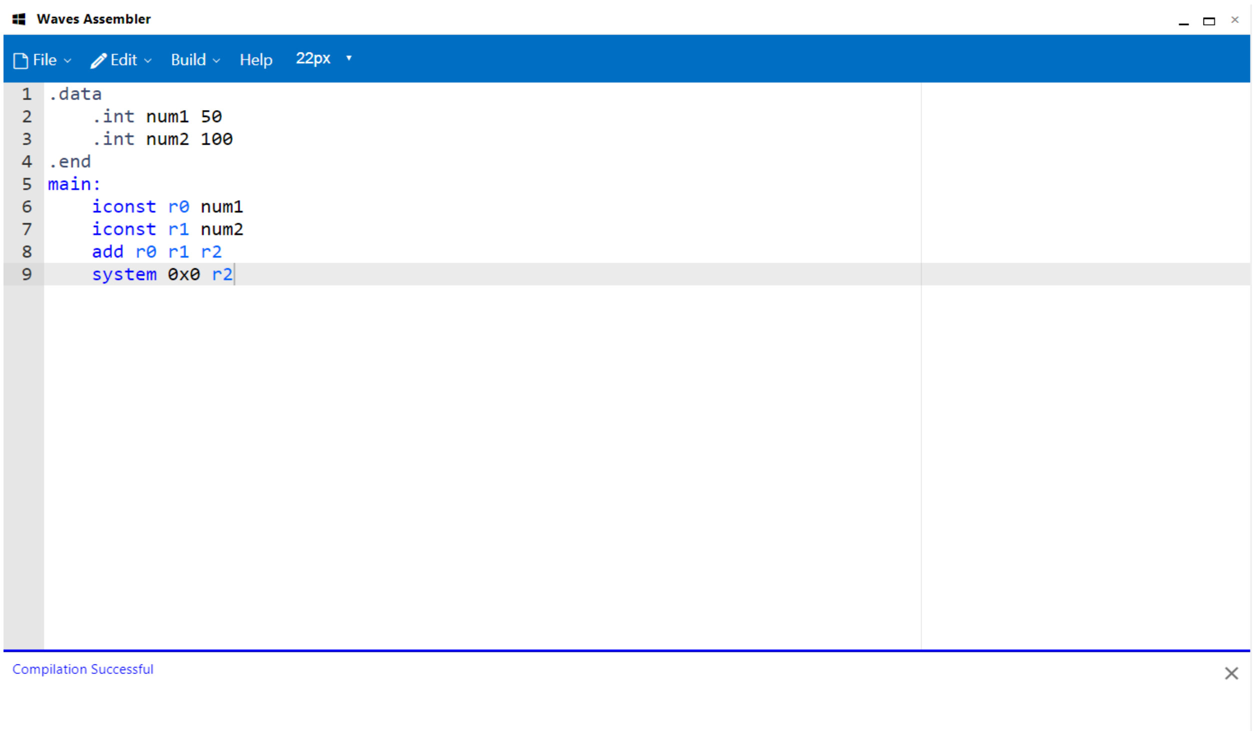
```
;.include "file.asm"
.data
    .string myStr "hello world"
    .int number 50
    .char character 'a'
.end
printString:

main:
    iconst r0 number   ;Move an integer constant number to r0
    add r0, r0, r1     ;r0=r0+r1
    sub r1, r2, r3     ;r1=r2-r3

    call printString  ;Call printString function
    system 0xf r3    ;Call the system function
.loop:
    goto .loop

    DD
```

```
Compilation Error
See log for details
```

```
Parse error on line 19:
...to .loop DD
----------------------^
Expecting ':', got 'EOF'
```

Figure 4.3 Compilation error popup window

Figure 4.3 above shows the compilation error popup window. This popup window is shown when users try to compile a program that has an invalid syntax.

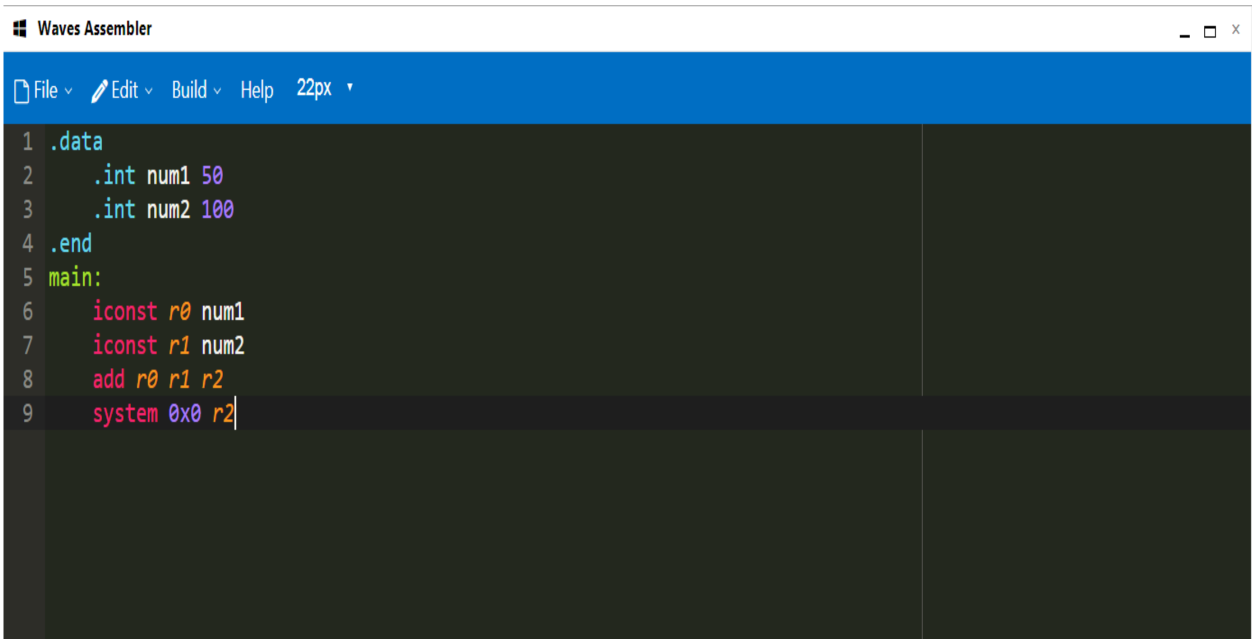


```
.data
    .int num1 50
    .int num2 100
.end
main:
    iconst r0 num1
    iconst r1 num2
    add r0 r1 r2
    system 0x0 r2
```

Compilation Successful

Figure 4.4 Compilation success popup window

Figure 4.4 above shows the compilation success popup window. This popup window is shown when a program has been successfully compiled by the user.

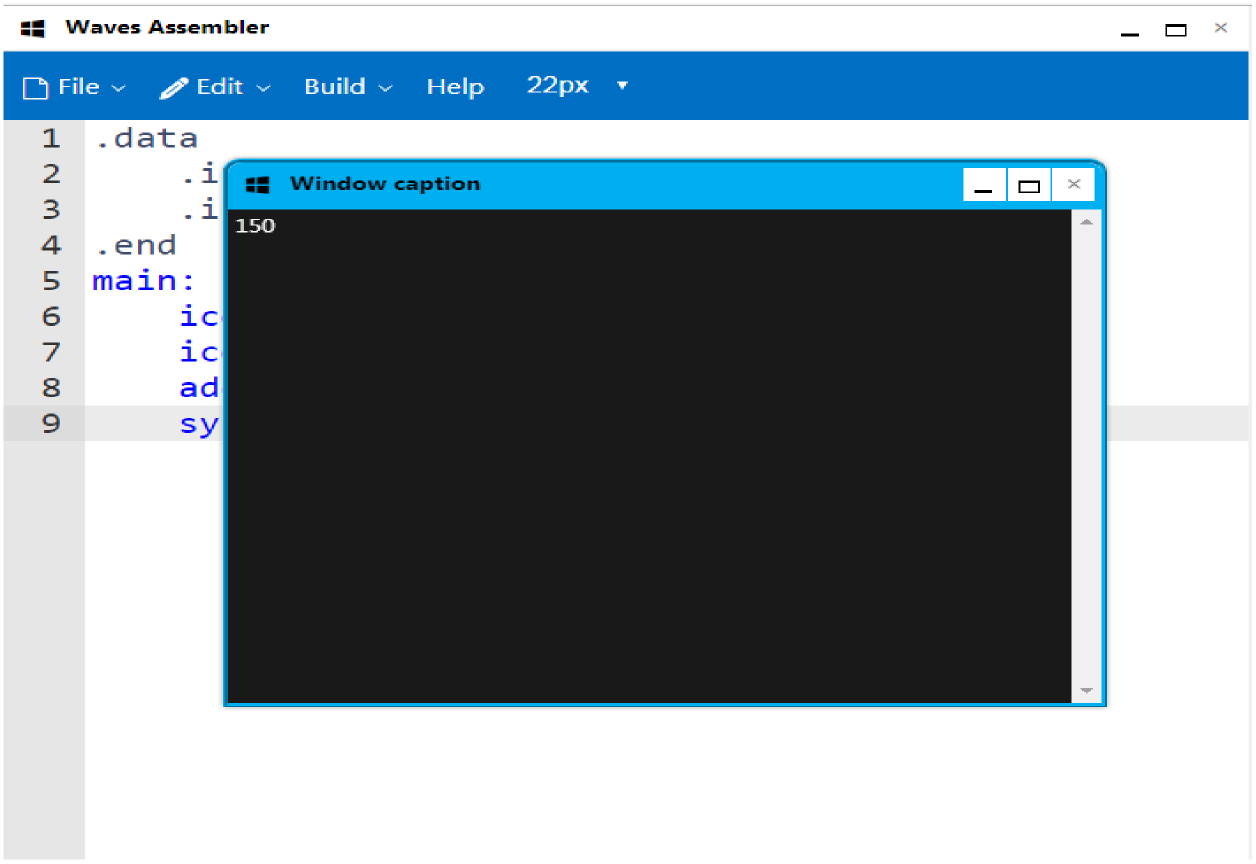


```
.data
    .int num1 50
    .int num2 100
.end
main:
    iconst r0 num1
    iconst r1 num2
    add r0 r1 r2
    system 0x0 r2
```

Figure 4.5 Changed Theme Interface

Figure 4.5 above shows the changed theme interface. This interface is shown after the theme has been modified by the user. Users can change the theme by clicking the edit button and then theme button.

4.4 Output Design

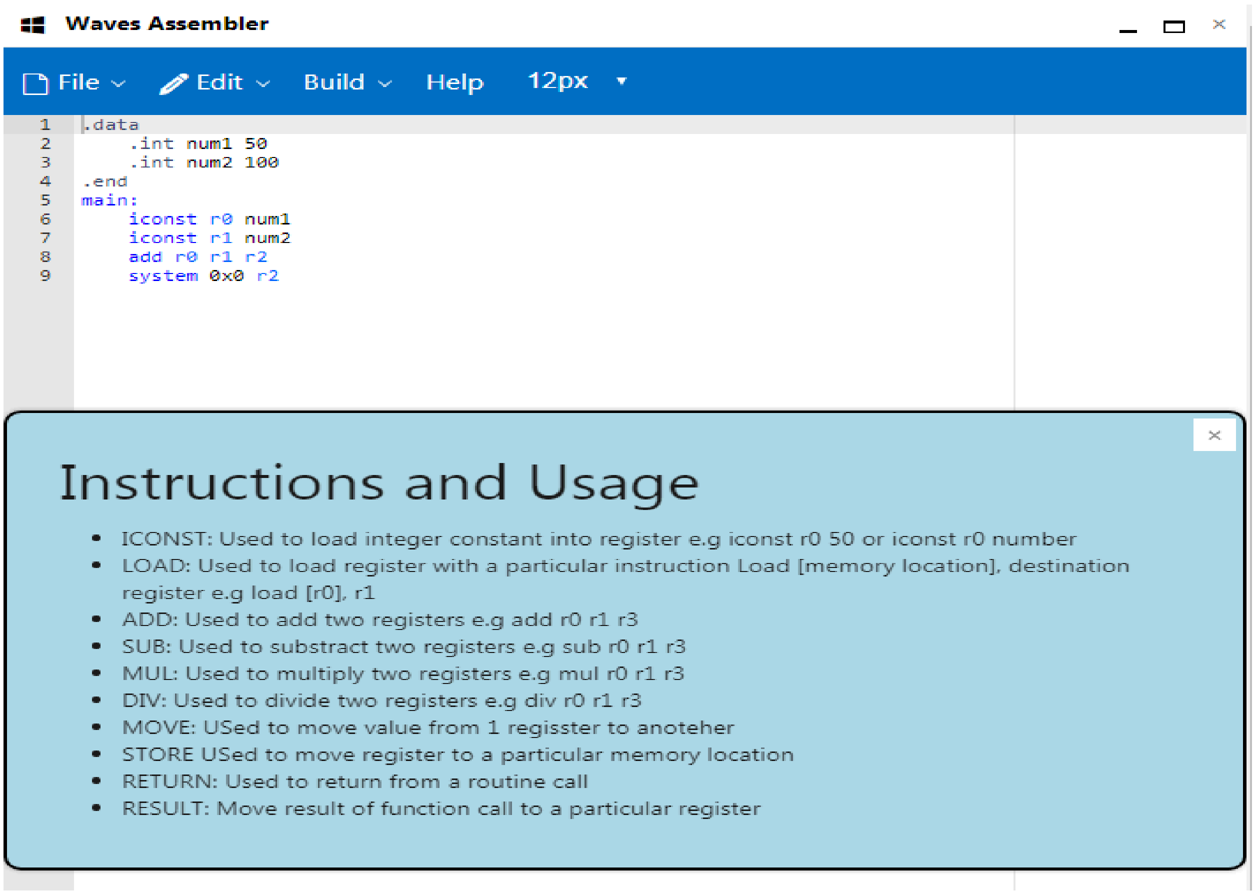


```
.data
      .i
      .i
.end
main:
      ic
      ic
      ad
      sy
```

```
150
```

Figure 4.6 Program output console window

Figure 4.6 above shows the program output console window. This console window will be shown after the run button has been clicked. It contains the output of the inputted program.



```
.data
     .int num1 50
     .int num2 100
.end
main:
     iconst r0 num1
     iconst r1 num2
     add r0 r1 r2
     system 0x0 r2
```

Instructions and Usage

- ICONST: Used to load integer constant into register e.g iconst r0 50 or iconst r0 number
- LOAD: Used to load register with a particular instruction Load [memory location], destination register e.g load [r0], r1
- ADD: Used to add two registers e.g add r0 r1 r3
- SUB: Used to substract two registers e.g sub r0 r1 r3
- MUL: Used to multiply two registers e.g mul r0 r1 r3
- DIV: Used to divide two registers e.g div r0 r1 r3
- MOVE: USed to move value from 1 regisster to anoteher
- STORE USed to move register to a particular memory location
- RETURN: Used to return from a routine call
- RESULT: Move result of function call to a particular register

Figure 4.7 Help Window

Figure 4.7 above shows the help window. This window will be shown after the help button has been clicked. It contains some help information on various instructions in Waves assembly language and how to use them.

CHAPTER FIVE

5.1 SUMMARY

The aim of this project is to develop and design Waves assembly language for web platform which aids beginner system programmers to learn assembly language on any device that has a web browser. Users can use Waves assembly language to start learning assembly language as Waves IDE has most of the features of a native IDE (Integrated Development Environment).

5.2 CONCLUSION

Assembly languages has always been a subject of universal interest, and the development of virtual machines has driven the requirement to produce fast and portable virtual computers that are capable of delivering speed close to that of a real computer. Virtual machines has uses in situations where user want to run more than one operating system on a single computer and also to provides a controlled environment for testing hardware that have not yet existed.

The proposed project has a platform by which new system programmers can get started in learning assembly language and also provides a controlled environment for testing the operation of assembly languages.

The developed system was found to satisfy, to large extent, the general requirements drawn for this project. Waves assembler was found to generate bytecode which can run in Waves virtual machine. Waves assembly language IDE was found to run on most web browser which supports JavaScript.

5.3 RECOMMENDATION

Although the developed assembly language is currently meeting the general requirements of the project and working fine as specified, there are still areas for modifications and future enhancement, this include the improvement in the performance of the system, making waves assembly language a native one and also for embedded system.

REFERENCES

Alexander, Dickson (2012). JavaScript Chip-8 Emulator. Retrieved from http://blog.alexanderdickson.com/javascript-chip-8-emulator.

Bieman, J.M. Murdock, V. (2001). Finding code on the World Wide Web: a preliminary investigation, Proceedings First IEEE International Workshop on Source Code Analysis and Manipulation.

Carlos, Rafael, Gimenes, das, Neves. (2013).

JavaScript assembly x86 compiler and emulator for educational purposes. Retrieved from http://carlosrafaelgn.com.br/Asm86

Howard, Raggatt. (1988). Arm assembly language. Retrieved from http://www.armarchitectures.com.au/people/howard-raggatt.

Jones, Derek (2009). Register vs. stack based VMs. Retrieved from http://shape-of-code.coding-guidelines.com/2009/09/register-vs-stackbased-vms/

Koetsier, Teun (2001). On the prehistory of programmable machines; musical automata, looms, calculators. PERGAMON, Mechanisma and Machine Theory 36.

Lindholm, Tim. Yellin, Frank. (1999). The Java Virtual Machine Specification. Second Edition. Sun Microsystems.

Paleczny, Michael. Vick, Christopher. Click, Cliff (2001). The Java Hotspot server compiler. Proceedings of the Java Virtual Machine Research and Technology Symposium on Java Virtual Machine Research and Technology Symposium Volume 1. Monterey, California: USENIX Association.

Pavone, Michael. Dex File Format.
Retrieved from http://www.retrodev.com/android/dexformat.html.

Radar oreilly. (2010). Counting programming languages. Retrieved from http://Radar.oreilly.com.

Rojas, Raúl, et al. (2000). The First High-Level Programming Language and its Implementation. Institut für Informatik, Freie Universität Berlin, Technical Report B-3/2000.

Saxon, James. Plette, William (1962). Programming the IBM 1401. Prentice-Hall. LoC 62-20615.

Security Engineering Research Group, Institute of Management Sciences. (2009).

Analysis of Dalvik Virtual Machine and ClassPath Library.

Smith, James. Nair, Ravi (2005). The Architecture of Virtual Machines. Computer. IEEE Computer Society. 38 (5): 32–38. doi:10.1109/MC.2005.173.

APPENDIX

INDEX.HTML

```
<!DOCTYPE html><html lang='en'><head> <link rel='stylesheet' href='lib/metro/css/metro.min.css'>
<link rel='stylesheet' href='lib/metro/css/metro-icons.min.css'> <link
href="lib/metro/css/metro-responsive.min.css" rel="stylesheet"> <link rel="stylesheet"
type="text/css" href="lib/metro/css/metro-colors.min.css"> <link rel='stylesheet' href='css/app.css'>
<meta name="viewport" content="width=device-width, initial-scale=1.0"></head><body> <div
class='window app'> <div class="window-caption"> <span class="window-caption-icon"><span
class="mif-windows"></span></span> <span class="window-caption-title">Waves
Assembler</span> <span class="btn-min"></span> <span class="btn-max"></span> <span
class="btn-close"></span> </div><div class="app-bar default" data-role="appbar"> <ul
class="app-bar-menu small-dropdown"> <li> <a href="" class="dropdown-toggle"><span class="icon
mif-file-empty"></span> File</a> <ul class="d-menu compact" data-role="dropdown"><!--<li><a
href="">New</a></li><li><a href="">Open</a></li>--> <li><a href=""
id="saveBtn">Save</a></li><li><a href="" id="closeBtn">Close</a></li></ul> </li><li> <a href=""
class="dropdown-toggle"><span class="icon mif-pencil"></span> Edit</a> <ul class="d-menu"
data-role="dropdown"> <li> <a href="" class="dropdown-toggle">Themes</a> <ul class="d-menu
themes-select" data-role="dropdown"> <li><a href=""
data-theme='ace/theme/monokai'>Monokai</a></li><li><a href=""
```

```
data-theme='ace/theme/eclipse'>Eclipse</a></li><li><a href=""
data-theme='ace/theme/gob'>Gob</a></li><li><a href=""
data-theme='ace/theme/dreamweaver'>Dreamweaver</a></li></ul> </li></ul> </li><li> <a href=""
class="dropdown-toggle">Build</a> <ul class="d-menu compact" data-role="dropdown"> <li><a
id="compileBtn">Compile</a></li><li><a id="runBtn">Run</a></li></ul> </li><li> </li><li><a
href="" id="helpBtn">Help</a></li><li> <select class='app-bar-element' id="fontSelect">
<option>10px</option> <option>11px</option> <option selected="true">12px</option>
<option>14px</option> <option>16px</option> <option>18px</option> <option>20px</option>
<option>22px</option> <option>24px</option> </select> </li></ul> </div><div
class="window-content" style="height: 100%"> <pre id="editor"></pre> </div></div></div><div
class="backdrop"> <div class="window" id="console" data-role='draggable'
data-drag-element='.window-caption' style="width:100%;max-width: 500px;"> <div
class="window-caption bg-blue"> <span class="window-caption-icon"><span
class="mif-windows"></span></span> <span class="window-caption-title">Window caption</span>
<span class="btn-min"></span> <span class="btn-max"></span> <span class="btn-close"></span>
</div><div class="window-content bg-dark fg-white"></div></div></div><div data-role="charm"
data-position="bottom" id='logger'> <div id="logger_message"></div></div><div data-role="dialog"
data-close-button="true" id="help"> <h1>Instructions and Usage</h1> <ul> <li>ICONST: Used to
load integer constant into register e.g iconst r0 50 or iconst r0 number</li><li>LOAD: Used to load
register with a particular instruction Load [memory location], destination register e.g load [r0],
r1</li><li>ADD: Used to add two registers e.g add r0 r1 r3</li><li>SUB: Used to substract two
registers e.g sub r0 r1 r3</li><li>MUL: Used to multiply two registers e.g mul r0 r1 r3</li><li>DIV:
Used to divide two registers e.g div r0 r1 r3</li><li>MOVE: USed to move value from 1 regisster to
anoteher</li><li>STORE USed to move register to a particular memory location</li><li>RETURN:
Used to return from a routine call</li><li>RESULT: Move result of function call to a particular
register</li></ul> </div><div data-role="dialog" data-close-button="true" id="save">
<h3>Successfully Saved file</h3> </div><script type="text/javascript"
src='js/jquery-2.1.3.min.js'></script> <script src="lib/ace/src-min/ace.js" type="text/javascript"
charset="utf-8"></script> <script src='lib/metro/js/metro.min.js'></script> <script
type="text/javascript" src='js/editor.js'></script> <script src='js/stringview.js'></script> <script
src='js/promise.min.js'></script> <script type="text/javascript" src='js/jsface.min.js'></script> <script
type="text/javascript" src='js/asm/Parser.js'></script> <script type="text/javascript"
src='js/asm/Utils.js'></script> <script type="text/javascript" src='js/FileSystem.js'></script> <script
type="text/javascript" src='js/asm/Assembler.js'></script> <script src='js/vm/vmUtil.js'></script>
<script src='js/vm/VM.js'></script> <script src='js/vm/OperatingSystem.js'></script> <script
type="text/javascript" src='js/Events.js'></script> <script type="text/javascript"
src='js/WavesEditor.js'></script> <script type="text/javascript" src='js/app.js'></script> <script
type="text/javascript" src='js/asm/test.js'></script> <script type="text/javascript"
src='js/store.legacy.min.js'></script></body></html>
```

APP.CSS

```
body,html{height: 100%; max-width: 100%;}.app{width: 100%; height: 100%
!important;}.app>*{}#editor{position: absolute; top: 0; right: 0; bottom: 0; left: 0; margin:
0;}.Warn{background-color: yellow;}select{border:0px;}#logger{height:100%;max-height:
200px;background-color: white;color:blue;border-top:3px ridge blue;}#help{padding:
20px;border-radius: 10px;border:2px solid black;background-color: lightblue;}.backdrop{z-index:
98;position: fixed;width:100%;height: 100%;top: 0;left:0;display: none;}#console{border:3px ridge
#00aff0;z-index: 100;position: absolute;top:10px;border-top-left-radius: 10px;border-top-right-radius:
10px;height: 100%;max-height: 400px;box-sizing:
border-box;}#console>.window-caption{border-top-left-radius: 8px;border-top-right-radius:
8px;border-bottom:2px solid #00aff0;box-sizing: border-box;}#console .window-content{height:
100%;max-height: 360px;overflow-y:scroll;white-space: pre-wrap;word-break: break-all;display:
pre;}
```

APP.JS

```
/** * @author Israel */"use strict";require('hello')Class(function(){var assembler; var editor; var
fileSystem; var virtualMachine; var operatingSystem; return{constructor: function(asm, ed, fs,
vm){assembler=asm; editor=ed; assembler.setFileSystem(fs); virtualMachine=vm;
operatingSystem=new OperatingSystem(); vm.setEvents(operatingSystem.events); this.init();}, init:
function(){var self=this; editor.onLoad(function(){});
editor.onCompile(function(source){assembler.assemble(source).then(function(res){se

lf.compiledBuffer=res.buffer; editor.compileSuccess();}, function(res){editor.compileError(res);
self.compiledBuffer=null;});});}); editor.onRun(function(){if
(self.compiledBuffer){virtualMachine.run(self.compiledBuffer);}else{editor.alert("Warn", "Source
file not compiled" );}});}, run: function(){}, main: function(App){$(document).ready(function(){var
asm=new Assembler(parser); asm.constructor(); var app=new App(asm, new WavesEditor, new
FileSystem('http://localhost'), new VM); app.run();});}};});
```

ASSEMBLER.JS

```
"use strict";var Assembler=function(){var fileSystem=null; var symbolTable=null; var
instrBuilder=null; function assembleData(obj){var promise=new Promise(function(resolve,
reject){obj.forEach(function(elem, index, obj){var data=obj[index]; try{var type; if
(data.type=="integer") type=SymbolTable.DATA_TYPE_INT; else if (data.type=="char")
type=SymbolTable.DATA_TYPE_CHAR; else if (data.type=="string")
type=SymbolTable.DATA_TYPE_STRING; symbolTable.setData(data.name, type, data.value);}catch
(e){throw{str: e.message, hash: data.info, type: Exception.DUPLICATE_DATA};}}); resolve();});
return promise;}function compile_routine(obj){var promise=new Promise(function(resolve,
reject){var totalSize=0; //Sum of the total size of all routines obj.forEach(function(elem, index,
obj){//List of the instructions var instrBody=[]; //Name of the routine var
routine_name=obj[index].label; var routineSize=0; var instructions=obj[index].value; if
(instructions==undefined) console.error(obj.length) //Try to Add a reference to instrBody to Routine
symbol table try{symbolTable.setRoutine(routine_name, instrBody, obj[index].info);}catch (e){var
error={str: e.message, hash: obj[index].info, type: Exception.DUPLICATE_ROUTINE};
reject(error);}for (var i=0; i < instructions.length; i++){var instr=instructions[i]; if
(instr.type=="program"){//Convert the instructions into numbers var
convertedInstr=instrBuilder.convertInstruction(instr); //Push converted instruction into the instrution
list instrBody.push(convertedInstr); //Increment the size of the routine routineSize
+=InstrBuilder.OP_MAP[instr.opcode].size;}else{//Try to set the local label
try{symbolTable.setLocalLabel(routine_name, instr.label, routineSize);}catch (e){var error={str:
e.message, hash: obj.info, type: Exception.DUPLICATE_LOCAL_LABEL};
reject(error);}}}symbolTable.setRoutineIndex(routine_name, totalSize); //Set index of routine into the
code totalSize +=routineSize; //Increment the total routine size
symbolTable.setRoutineSize(routine_name, routineSize); //this.InstrBuilder.OP_MAP[name];
//console.log(symbolTable.getRoutine(routine_name))}); resolve();}); return promise;}function
handleIncludes(includes){var promise=new Promise(function(resolve, reject){for (var i=0; i <
includes.length; i++){var inc=includes[i];
this.compile_file(inc.value).then(function(res){resolve(res);}, function(e){reject(e);});}}.bind(this));
return promise;}//Build all routines in instruction and partially build constants function
buildRoutine(code){//The array to store binary code var code=[]; //The size of the routines in byte var
size=0; //Increase the size of the instructions function incSize(s){size +=s;}//Add the routine index
into the code function addRoutineIndex(name){}//Get all available routine names var
allRoutines=symbolTable.getAllRoutineNames(); allRoutines.forEach(function(elem, index,
obj){//Loop over the routine names to convert it to binary //The first routine var
name=allRoutines[index]; //console.log(symbolTable.getRoutineSize(name)); //The data of of the
routine var routine=symbolTable.getRoutine(name); //The instructions in the routine var
instructions=routine.value; //Loop over the instructions to convert them
instructions.forEach(function(elem, index, instructions){var instr=instructions[index];
//code=code.concat(instr); var opcode=instr[0]; //First Value as opcode switch (opcode){case
InstrBuilder.Opcodes.ADD_OP.code: case InstrBuilder.Opcodes.SUB_OP.code: case
InstrBuilder.Opcodes.MUL_OP.code: case
InstrBuilder.Opcodes.DIV_OP.code:{code=code.concat(instr); break;}case
InstrBuilder.Opcodes.GOTO_OP.code:{var opcode=instr[0]; var localLabel=instr[1]; if
(symbolTable.localLabelExist(name, localLabel.name)){//The index of local label into the routine var
index=symbolTable.getLocalLabel(name, localLabel.name); code=code.concat([opcode,
index]);}else{var error={str: "Local label does not exist", hash: localLabel.info, type:
Exception.UNKNOWN_LOCAL_LABEL}; throw error;}break;}case
```

```
InstrBuilder.Opcodes.ICONST_OP.code:{var opcode=instr[0]; var reg=instr[1]; var value=instr[2]; if
(value instanceof Object){//If it is an identifier if (symbolTable.exists(value.name,
SymbolTable.TYPE_DATA)){var index=symbolTable.getDataIndex(value.name); var start=index >>
8; var end=index & 0x00ff; code=code.concat([opcode, reg, start, end]);
//console.error(symbolTable.getAllData());}else{var error={str: "Data does not exist", hash:
value.info, type: Exception.UNKNOWN_DATA_DFN}; throw error;}}else{//If it is of type integer
var dataName=symbolTable.newSystemData(value, SymbolTable.DATA_TYPE_INT); var
index=symbolTable.getDataIndex(dataName); var start=index >> 8; var end=index & 0x00ff;
code=code.concat([opcode, reg, start, end]);}break;}
```