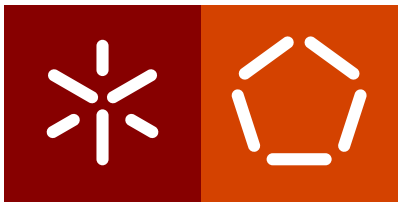
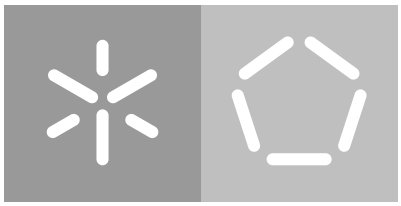


Universidade do Minho
Escola de Engenharia
Departamento de Informática

Sofia Almeida Teixeira

EWVM - an Educational Web Virtual Machine

November 2022



Universidade do Minho
Escola de Engenharia
Departamento de Informática

Sofia Almeida Teixeira

EWVM - an Educational Web Virtual Machine

Master dissertation
Integrated Master's in Informatics Engineering

Dissertation supervised by
José Carlos Ramalho
Pedro Rangel Henriques

November 2022

STATEMENT OF INTEGRITY

I hereby declare having conducted this academic work with integrity. I confirm that I have not used plagiarism or any form of undue use of information or falsification of results along the process leading to its elaboration.

I further declare that I have fully acknowledged the Code of Ethical Conduct of the University of Minho.

Sofia Teixeira

_____

ABSTRACT

The Language Processing Course at Minho's University uses a Virtual Machine implemented in C with its interface being implemented with the GTK toolkit. However, it is neither very informative nor very easy to install.

The goal in this Master's Project is to analyze and model the entire Virtual Machine's system and build a Web application with a graphical interface. The new tool offers two main characteristics: compiling and reporting errors in programs written for the Virtual Machine; and animate its execution, displaying the internal state of the VM and providing the user an interface to control the execution.

In this document, a study of existing technologies will be carried out, focusing in detail on the current virtual machine VM. After this analysis, a solution will be proposed, followed by a detailed explanation of its implementation.

Keywords: Virtual Machines, Stack Machines, Register Machines, Compilers, Assemblers

RESUMO

Na Unidade Curricular de Processamento de Linguagens tem-se utilizado uma VM doméstica implementada em C com uma interface GTK. No entanto, esta não é muito informativa nem muito fácil de instalar.

O objetivo nesta dissertação é fazer uma análise e modelação de todo o sistema e construir uma aplicação Web com uma interface gráfica. A nova ferramenta oferece duas funcionalidades principais: compilar e reportar erros em programas escritos para a VM e, se o programa estiver correto, animar a sua execução mostrando o estado interno da VM e fornecendo ao utilizador uma interface de controlo sobre a execução.

Neste documento, será realizado um estudo das tecnologias existentes, focando em detalhe a máquina virtual atual VM. Após esta análise, será apresentada uma proposta de solução, seguida de uma explicação detalhada da sua implementação.

Palavras-chave: Máquinas Virtuais, Máquinas de Stack, Máquinas de Registos, Compiladores, Assemblers

CONTENTS

LIST OF FIGURES

1

INTRODUCTION

A Virtual Machine (VM) is a software layer over the real machine. A VM creates a virtualized environment that mimics a computer system. It behaves like a completely separate computer, running independently from the actual computer (host) and other virtual machines (guests). Each VM runs its own operating system and functions, also having virtual hardware. Allowing multiple operative systems to run in a single physical computer is a very relevant feature of a virtual machine (IBM Cloud Education, 2019).

Many virtualization solutions have been implemented for various intents. In cloud environments, VMs are fundamental since they provide virtual application resources to multiple users at once. Virtual Machines are also demanded for security purposes, since they allow the user to take risks that could otherwise harm the computer, making them great for things like malware analysis (Nguyen et al., 2009). The most obvious use for VMs is to run incompatible software, as is the need to test new operating systems. Some VMs, like Java Virtual Machine (Subhashana, 2021), were created to assure software compatibility, allowing all operating systems to equally run the same source code, previously compiled.

A growing use for Virtual Machines is in the field of education, since these allow for a more specialized environment. There has been a focus on virtual machines being used with pedagogical purpose (AlNajdi et al., 2020), such as teaching various coding fields, namely cybersecurity (Chothia and Novakovic, 2015), system software development (Driscoll et al., 2004) and others (Hu and Han, 2009).

1.1 MOTIVATION

At the moment, a Virtual Machine, named VM, developed by Jean-Christophe Filliatre at LRI/Université de Paris/Saclay for his Compiler courses (de Sousa, 2006), is used in the Language Processing Course to teach students about compilers and code generation. The current Virtual Machine is implemented in C and it runs programs written by the programmers in VM-Assembly language. As opposed to register based virtual machines, this one operates mainly using stacks, relying on only a few control registers (Shi et al., 2008) like the Stack-Pointer (SP), the Program-Counter (PC) and the Global and Frame Pointers (GP and FP). A Stack-Machine requires that all the operands of any arithmetic, logic, load/store or input/output operator are located at the top of the stack; after pop them and execute the operation, the result is then pushed into the stack. The interaction between the user and the virtual machine is conducted through the computer's terminal. The user-machine's communication is limited to what the given program is coded to do, so there is no additional information shared about how the machine works and its internal state.

The main motivation to create this new Virtual Machine relates to the difficulty installing the current one. With this in mind, this Master's work will focus on developing a new virtual machine that will replicate the old one, but without the need to be installed. Furthermore, the goal is also to improve it, aiming to achieve a more user friendly interface that is also more informative.

To that end, this project consists in developing a Web Application, which runs the Virtual Machine at the server side. To improve the user's understanding of the machine, it will have a visualization option of its state while it processes the given program.

1.2 OBJECTIVES

This Master project's goal is to implement a Virtual Machine running on a Web Application. To accomplish this, it will be necessary to:

- Develop a Grammar to define the Assembly Language;
- Develop the new Virtual Machine;
- Develop a Web Interface;
- Extend the new machine pedagogical functionalities.

1.3 RESEARCH HYPOTHESIS

By the end of this Master's Project it will be established that by using a Web approach it is possible to have a more accessible Virtual Machine.

1.4 DEVELOPMENT APPROACH

The methodology that will be followed in this Master's thesis is composed of the following steps:

- Bibliographic search to find documents concerning virtual machine and their implementation approaches and techniques;
- Reading and synthesis of the bibliography selected;
- Propose a solution;
- Develop the solution;
- Test and discuss results;
- Extension of the instruction set and optimization of the VM performance.

1.5 DOCUMENT STRUCTURE

Throughout this document, various topics related to this project will be discussed.

Chapter 2 analyses and assesses the existent technology related to the subject-matter of this Master's Project.

Chapter 3 focuses on the currently used Virtual Machine, VM, which provided inspiration for this project.

Chapter 4 presents a detailed description of the proposed solution for this project and what features it aims to achieve.

Chapter 5 describes the development and functioning of the Assembler and its role in the Web-based software system.

Chapter 6 explains the structure of the Virtual Machine.

Chapter 7 focuses on the organization of the GUI explaining how the different components interact with each other and how to navigate them.

Chapter 8 presents a conclusion that summarizes all the information presented along the report and emphasizes the work done, reflecting on what was achieved, as well as proposing directions for future work.

2

STATE OF THE ART

In the 1960s, Input/Output processors and multiprogramming emerged, meaning that multiprocessors could have access to a common main memory and that multiple processes could share a single processor and compete for a common pool of resources. Since one cannot trust all the software to be correct, this created integrity problems. An I/O processor could alter the main memory areas related to another processor, consequently wrongly modifying such processor's execution. In the same way, a process could alter resources related to another process, consequently wrongly modifying the process execution. This study was based on Buzen and Gagliardi's study (Buzen and Gagliardi, 1973).

The solution to these problems was to create a **Dual State Architecture** (see Figure 1). This consists in dividing the software in two parts:

- **privileged/system software:** encloses a small part of code that runs critical operations
- **non-privileged/user software:** is composed by the rest of the code in which processes are not permitted to interfere with each other

As such, the privileged code would interact directly with the basic machine interface and would build the extended machines as non-privileged code, where the user would program.

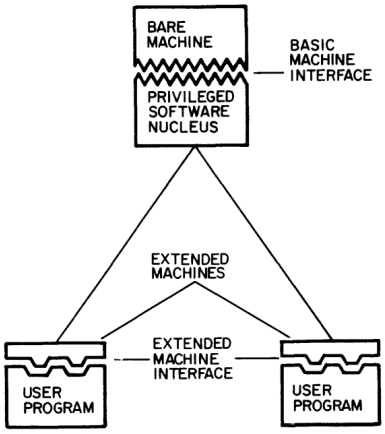


Figure 1: Dual State's Architecture.
Image taken from (Buzen and Gagliardi, 1973)

Nevertheless, this solution did not solve everything. Since the extended machines are non-privileged code, programs written for one cannot be ran in another, creating practical difficulties for the users.

On top of that, given that only the privileged software can access and control the hardware functionalities, it creates a few drawbacks regarding its use. Considering that, with this architecture, it is impossible to have one than more version of the privileged part, only one programmer can access it at time, which significantly slows the development process.

In addition, tests and diagnosis need to be carried out with hardware access. This causes issues with the usual functionalities of the privileged code since both cannot run at the same time.

Since most issues come from only having one privileged software nucleus, the key solution would be to create more. If the Extended Machine Interfaces could be replaced by copies of the Basic Machine Interface, each could run its own privileged nucleus. Accordingly, the conclusion was that having multiple basic machine interfaces would play a pivotal role in fixing the referred problems.

In actuality, when a basic machine interface is not supported directly on a Bare Machine, it is called a **Virtual Machine**.

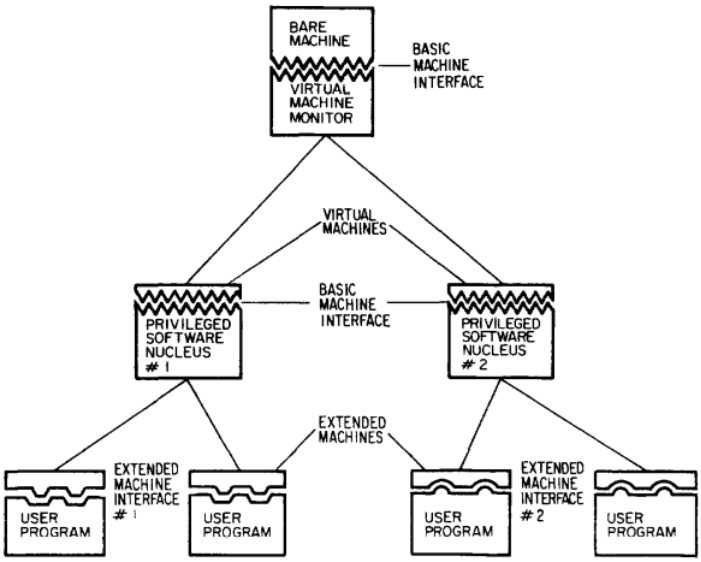


Figure 2: Virtual Machine's Architecture - Type I.
Image taken from (Buzen and Gagliardi, 1973)

The basic machine interfaces supported by the **Virtual Machine Monitor (VMM)** and by the Bare Machine are identical, therefore the same privileged software can run on both, performing with no knowledge on where it is running. In this manner, a Virtual Machine is fundamentally equivalent to a real machine.

Nonetheless, they are not the same. The most impactful difference rests in performance time. The Virtual Machine will always take longer due to VMM overhead delays and time access to a busy engaged single processor. In addition, Virtual Machines sometimes lack minor functional capabilities like being able to run self modifying programs.

Efficiently, for most of the time the Virtual Machine Monitor allows the programs to run directly on the Bare Machine, only stepping in to secure the integrity of the system, trapping and executing specific instructions.

The architecture of the **Virtual Machine Type I** (see Figure 2) was not the only Virtual Machines resolution regarding the problems described. The basic machine interface supported by the Bare Machine does not need to be similar to the one supported by the Virtual Machine Monitor. This is called an **Emulator** and gives rise to a **Virtual Machine Type II** (see Figure 3). It is possible to create VMMs that also run on Extended Machines.

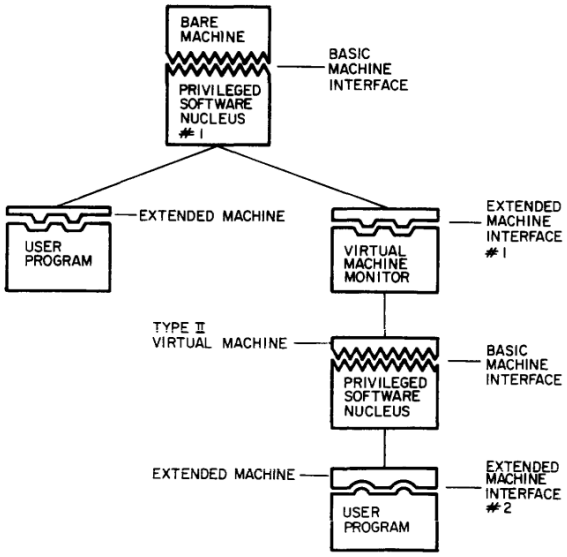


Figure 3: Virtual Machine's Architecture - Type II.
Image taken from (Buzen and Gagliardi, 1973)

This type of Virtual Machine comes with a few advantages. Virtual Machines Type I deal with problems of recursion due to the use of similar interfaces. In opposition, Emulators charts its basic machine interface onto the extended machine interface, not facing the same problem. Some other issues the Emulators evade are the scheduling and allocation problems that come with operating multiple independents copies of the basic machine interface.

However, the Emulators do tackle more complex I/O problems since the emulated system and its extended machine are running on different I/O devices and channel architectures.

2.1 STACK VS REGISTER VIRTUAL MACHINES

Within the Virtual Machine, the current technologies are constructed by one of the two prevalent architectures up-to-date. Effectively, present-day Virtual Machines are either register based or stack based.

A **Stack Virtual Machine** uses a stack data structure as memory. It works by pushing and popping values from the top of the stack. To execute these movements, the machine relies on the stack pointer (SP), which points to the top of the stack at all times. Any instruction that requires operand values to be performed pops them from the top of the working stack. Then the operation is executed outside of the stack and its result is pushed back to the stack (Sinnathamby, 2012).

Example: If one wanted to add two values and store the result, there are several steps that need to be taken (see Figure 4):

1. **PUSH V1 -** Puts value 1 on top of the stack
2. **PUSH V2 -** Puts value 2 on top of the stack
3. Add the values and store the result:
   a) **POP -** Takes value from the top of the stack
   b) **POP -** Takes value from the top of the stack
   c) **ADD -** Adds the values
   d) **PUSH R -** Puts result on top of the stack

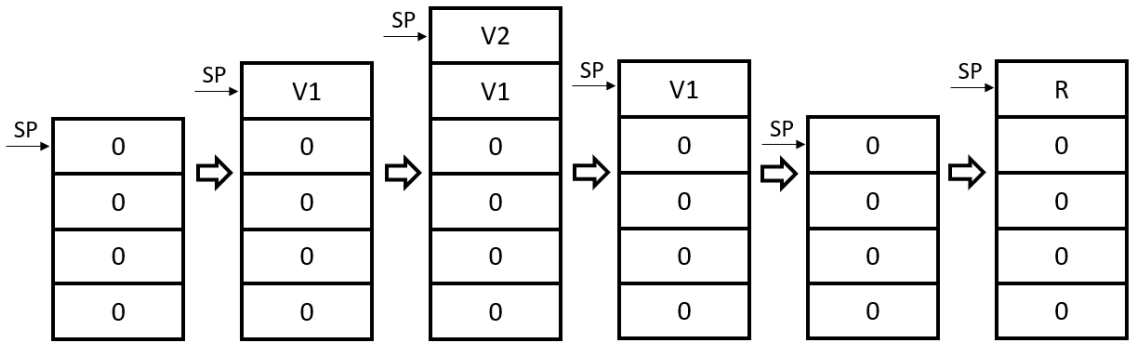


Figure 4: Stack Example Illustration.
Image inspired by Sinnathamby (2012)

A **Register Virtual Machine** stores its data in registers of the CPU. The instructions must indicate the registers used to store the operands required in each case. It works faster than a stack based virtual machine within the instruction dispatch loop in the sense that it avoids the overhead of all the popping and pushing that would be otherwise necessary Sinnathamby (2012). However that more efficient approach requires from the compiler an extra task that is complex: the registers management.

Example: If one wanted to add two values and store the result, there are several steps that need to be taken (see Figure 5):

1. **PUT V1 R1 -** Puts value 1 on register 1
2. **PUT V2 R2 -** Puts value 2 on register 2
3. **ADD R1 R2 RX -** Adds the values from R1 and R2 and puts the result in register RX

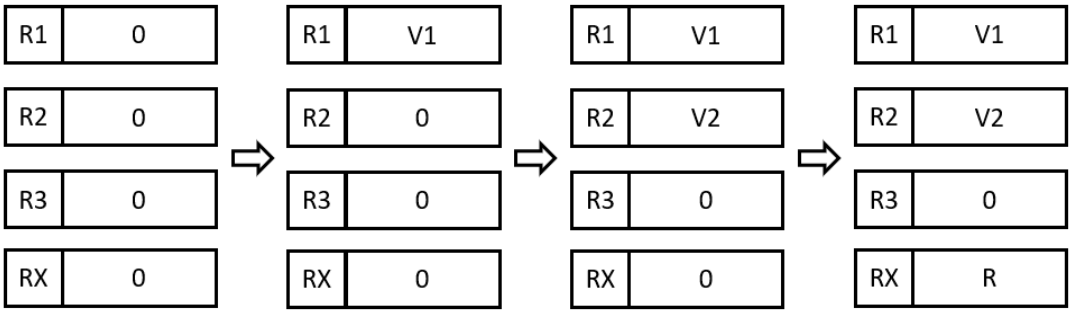


Figure 5: Register Example Illustration.
Image inspired by Sinnathamby (2012)

Which type of Virtual Machine is better seems to be a debatable subject. Interestingly, there is a research paper where the authors rewrite the Java Virtual Machine as a register based Virtual Machine Shi et al. (2008) and accomplish a superior performance.

However, Stack Virtual Machines are more attractive as a Learning Resource considering that the operands location is implicit in the stack pointer, unlike register based machines where it needs to be specified Davis et al. (2003). Consequently, stack based machines also tend to run a simpler and easier to comprehend instruction set, a valuable feature in terms of execution and application Kexugit (2011), though it must be noted that this feature often results in more extensive programs.

Although Register Virtual Machines seem to be able to attain a higher performance, this does not seem to be easily achievable. Their more complex and extended instructions require a broader instruction set, raising the interpretation difficulty. For a further discussion about how to deal with register based instructions, see the Dragon book Aho et al. (1986).

2.2 VIRTUALIZATION

Effectively, Virtual Machines are the embodiment of virtualization. Virtualization is when a simulated computing environment is created instead of a physical version.

This technology brings tremendous progress and practicability to the everyday life of programmers. Whereas in previous times, one would have multiple physical machines for different requirements and needs, in these days, one can implement all those distinct machines with different features in a single computer.

There are five main types of virtualization (Baca, 2021):

- **Server virtualization:** Multiple virtual versions of computers and operating systems (VMs) are run in a single physical server. Each of the versions is managed by a Virtual Machine Monitor, which allocates and distributes resources, improving their usage efficiency.
- **Application virtualization:** An application is implemented on a single computer system and its access is available to other people, within specified permissions. As such, the users can operate remote software not installed on their machine.
- **Desktop virtualization:** The application and operative system are installed on a virtual machine that runs on a remote server. This setup allows people to work on a different system than their machine's, possibly sharing it with other people, making it easier to share information.
- **Storage virtualization:** Multiple physical storages are combined into a virtual storage system, functioning as if they were one single storage device. Because transporting and backing up data is significantly easier with this sort of virtualization, it is widely utilized for disaster recovery.
- **Network virtualization:** Multiple sub-networks are created on the same physical network. Each of these individual channels can be assigned to users, servers or devices. Since it is a simulated network, it is perfect for testing new programs and apps before their release to the public.

Virtualization seems to be highly advantageous not only for programmers and the IT world but also for businesses and companies. As such, the focus on Virtual Machines seems to have passed from compilers to cloud computing and such types of online virtualization.

2.3 CURRENT VIRTUAL MACHINES

Over the years, besides the more straightforward Virtual Machines based on the initially explored matters, there are some VMs that were born from specific needs and problems. Kohlbrenner et al. discusses a few of these Virtual Machines.

In this section, some of those Virtual Machines and models will also be looked into alongside a few others.

2.3.1 *IBM Virtual Machine*

> *The IBM model solved the need for a single computer system that could simulate multiple computers with different operating systems.* (Kohlbrenner et al.)

The **IBM model** works by creating multiple copies of the original machine. It divides its memory and resources and assigns them to each duplicate. Every single one of these copies is a Virtual Machine, so it functions isolated from original machine and from the others.

Each VM behaves exactly like the original machine but with lesser memory capacity. Every instruction programmed is directed to an equivalent machine instruction and executed as such, so the user is allowed to make whichever operations wanted just as he would in the original machine. This, however, may cause security issues. If the user has permission for every possible operation, it means they are allowed to change others or the original machine's state or even change some important secured machine functions.

To maintain systems integrity, sensitive instructions that could cause these problems are segregated and dealt with in a different separate manner. Instead of executing these directly, the VM simulates its functioning. This is done in a way that the user does not realize they have written a problematic instruction that was executed differently.

IBM Cloud allows the user to create their own Virtual Machine in which they may choose its features, such as memory, local storage and GPU (IBM Cloud Education, 2019). These Virtual Machines are created and work based on the previously described model.

Cloud computing makes computer resources available for use through the internet. Private Cloud is a cloud computing environment destined to be used by a single entity. As such, its hardware and software cannot be accessed others. **IBM Cloud Private** runs on public clouds, one of which is the **Microsoft Azure**.

Azure is a cloud computing platform that provides access, through an online portal, to Microsoft's services and resources, such as storing and transforming data (Simplilearn, 2021). Its simplified architecture (see Figure 6) is based on what was previously described, having the storage divided through its VMs and a Hypervisor that oversees the instructions.

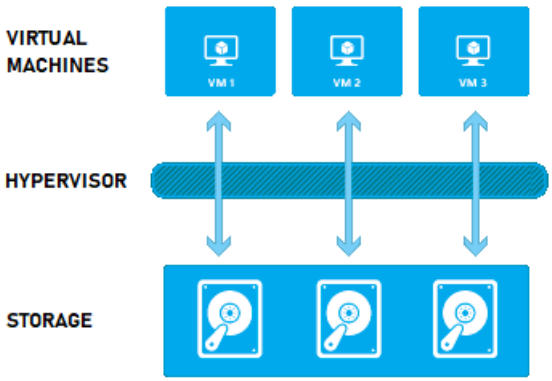


Figure 6: Azure Architecture.
Image modified from (Virdee, 2015).

2.3.2 *Java Virtual Machine*

> *The Java Virtual Machine addresses the need for an application that will run on multiple operating systems.* (Kohlbrenner et al.)

The Java language was built on a Virtual Machine. The Java Virtual Machine (JVM) runs on top of the computer's Operative System (OS). The JVM has its own compiler and interpreter that works for each OS (see Figure 7). It is implemented this way so that the programmer communicates directly with it, writing code with no need to take notice of its OS particularities. This allows for portability since it means the same exact code can be executed in multiple platforms. The Java Virtual Machine is also quite small which facilitates its use and application. However, JVM has not reached full portability since it is still not fully adapted to all possible Operative Systems.

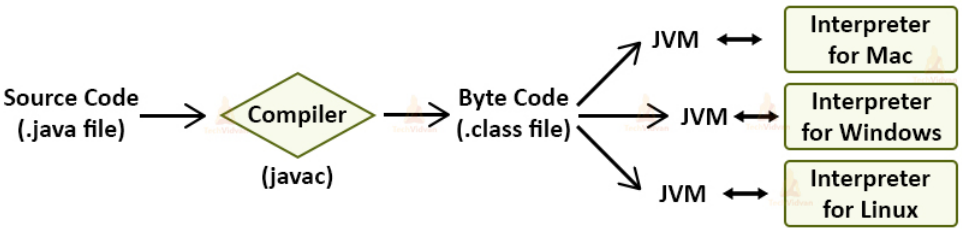


Figure 7: JVM - Multiple Interpreters.
Image taken from (TechVidvan, 2021).

Java programs start in a java file written by the programmer. This file goes through a compiler which generates a class file by turning the java code into java bytecodes. By this point the file is ready to be interpreted and executed by the Java Virtual Machine. All libraries needed are also imported to the JVM to help executing the program. These steps are depicted in Figure 8.

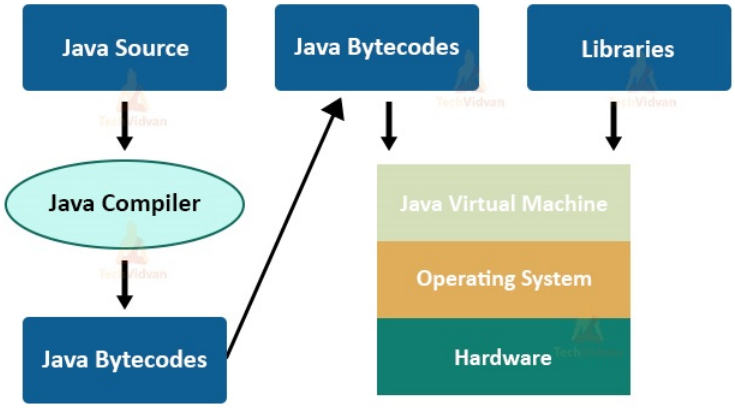


Figure 8: Java - Architecture.
Image taken from (TechVidvan, 2021).

The Java system is composed of three environments, as illustrated on Figure 9.

- **JVM - Java Virtual Machine:** to execute code
- **JDK - Java Development Kit:** to run a program
- **JRE - Java Runtime Environment:** to develop a program

Figure 9: Java Platform Components.
Image taken from (TechVidvan, 2021)

.

JVM Architecture

The Java Virtual Machine is divided in four main constituents (see Figure 10). Each component is broken down into several others sectors that will play a substantial part in the machine's functioning.

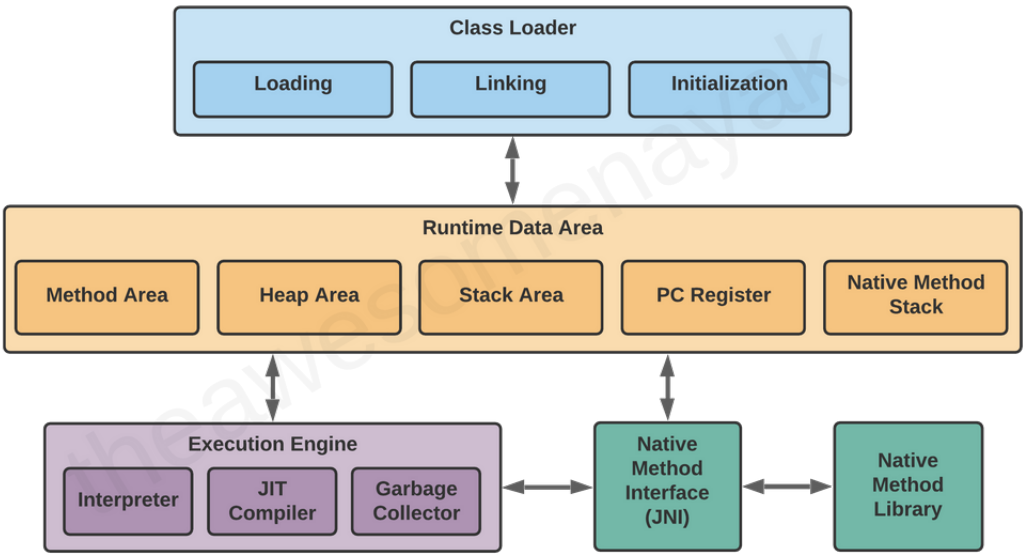


Figure 10: JVM Architecture.
Image taken from (Nayak, 2021).

Class Loader

When the class files are sent to the JVM, these are received by the Class Loader. This component is responsible for loading and processing them. They undergo three stages.

1. Loading

In the loading stage, the bytecode is withdrawn from the class files and saved in the Method Area. With this data, JVM generates an Class type object and saves it in the Heap Area.

2. Linking

In the linking stage, a verification is ran in which the system checks if the class files are valid and correct. If all goes well, it allocates variables and initializes memory default values. At last, the machine replaces symbolic references with direct references.

3. Initialization

In the initialization stage, the static variables are assigned their defined values.

Runtime Data Area

This component is the memory area of the Virtual Machine. It holds several sections for the storage of different types of data.

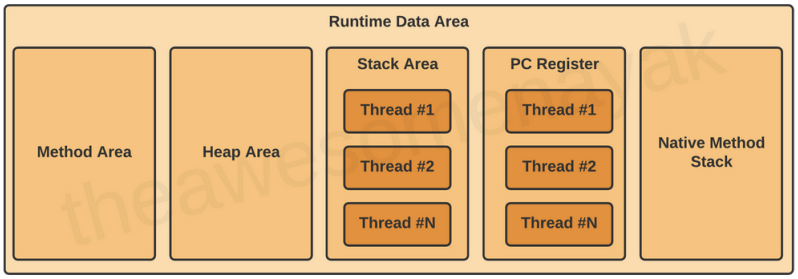


Figure 11: JVM Runtime Data Area Architecture.
Image taken from (Nayak, 2021).

The Runtime Data Area is divided in several other areas (see Figure 11):

1. Method Area: It stores the class data and the code for methods and constructors.

2. Heap Area: It is the runtime area that stores objects and their variables.

3. Stack Area: It stores the local variables of a program. Whenever a new thread is created, another runtime stack is generated.

4. Program Counter Register: It stores the current instruction's address. Each thread has its own register.

5. Native Method Stack: It contains the native methods, not written in Java, required in the application.

Execution Engine

The Execution Engine is the section that actually runs the program. It accesses the data from the memory and executes the instructions.

1. Interpreter

The Interpreter reads the bytecode and executes the instructions. This is done by going through instructions one by one, which does not make it a speedy process. Repeated called methods are treated the same as new methods, which is not very efficient.

2. Just-In-Time Compiler

The JIT Compiler saves the already called methods and their corresponding machine instructions in cache. While compiling the code, if there is code with similar functionality, it gets its machine code from cache. In this manner, the interpreter's performance is balanced out, improving the system's performance.

3. Garbage Collector

The Garbage Collector removes unreferenced objects from the heap area in order to regain unused memory.

Native Method interface

Native Method interface is a framework that promotes communication between different languages. It also allows Java code to can call libraries and native applications.

2.3.3 *Google Colab Virtual Machine*

Google Colab is a very popular Virtual Machine within the Machine Learning and Data Analysis areas. It is designed for programmers who need more GPU power for their work. Effectively, Google Colab allows the usage of cloud GPU for free, providing a much better performance than the average user's computer.

2.3.4 *Parrot Virtual Machine*

Parrot is a register-based Virtual Machine that efficiently runs dynamically typed languages. These are languages in which the interpreter assigns a type to variables at run-time. Parrot was created to implement Perl6's language. Nonetheless, it was designed to be able to execute various other languages, like Python or Ruby.

There are four components to Parrot's architecture (see Figure 12): Parser, Compiler, Optimizer and Interpreter. Based on the source code, the Parser creates an Abstract Syntax Tree (AST), which is sent to the Compiler. It is then turned into bytecode and it is either sent to the Optimizer or directly to the Interpreter. The Optimizer takes the bytecode and the abstract syntax tree, if necessary, and attempts to achieve the best possible optimized bytecode. Finally, the Interpreter runs the code (Fagerholm, 2005).

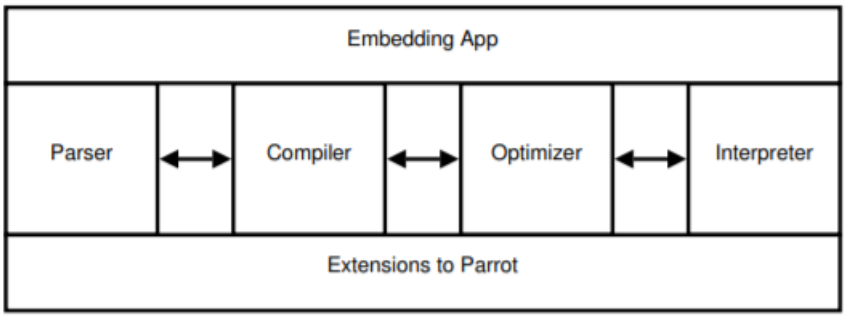


Figure 12: Parrot's System Architecture.
Image taken from (Fagerholm, 2005).

Traditionally, languages are either interpreted from their source code, compiled directly to machine code or compiled to bytecode and then interpreted. Parrot can accept a program's source code and parse and compile it to bytecode at run-time. As a result, execution will start with compiling the code to bytecode and then proceed to execute it.

Rather than executing the code, it is possible to replace Parrot's interpreter with one that saves the bytecode into a file. As a result, this allows for the saved bytecode to be executed independently without having to go through parsing, compilation and optimization processes.

Parrot's interpreter also includes the option to transform Parrot's bytecode into another language's bytecode. For an extended explanation of all Parrot's features see (Fagerholm, 2005).

2.3.5 *Python Virtual Machine*

Python is an interpreted high-level object-oriented programming language. Python programs are cross-platform, so they be run on every platform without having to be rewritten (K, 2021). Python's interpreter transforms source code into machine code. There are a few interpreters implementations for python:

- CPython: a Python interpreter written in C language
- PyPy: a Python interpreter with Just-in-time (JIT) compilation
- Jython: an implementation designed to run on the Java Platform
- IronPython: an implementation targeting the .NET Framework

CPython is the Python language's default and the most extensively used implementation, so the next information will focus on its functioning.

The execution of a Python program with CPython is divided into two or three main stages (Ike-Nwosu), as depicted in Figure 13.

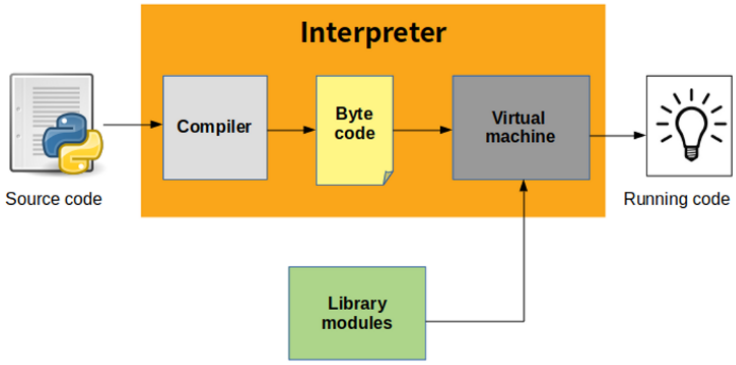


Figure 13: Python's System Architecture.
Image taken from (K, 2021).

1. **Initialization:** This stage is only significant when a program is ran non-interactively through the command prompt. It handles the preparation of data structures, memory allocation and other settings required by the Python process.

2. **Compilation:** This stage is responsible for the several steps entailed in the process of turning source code to bytecode. These steps can be observed in Figure 14.

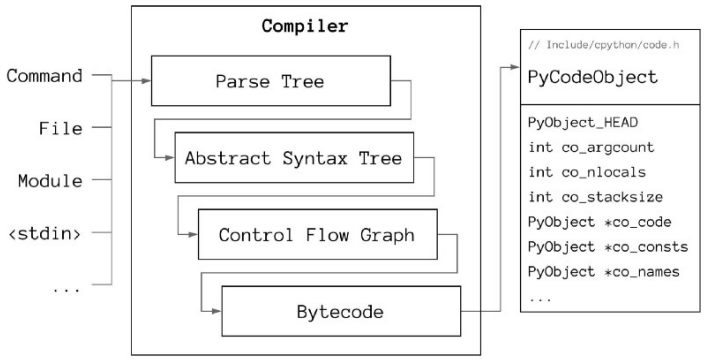


Figure 14: CPython's Compiler.
Image taken from (Prantl, 2020).

a) From Source To Parse Tree

The Python parser is an LL(1) parser based on the Dragon book's (Aho et al., 1986) description of such parsers. The parser tokenizes the source code and arranges the tokens as nodes in a Parse Tree.

b) From Parse Tree to Abstract Syntax Tree

The system goes through the Parse Tree nodes and transforms them into corresponding Abstract Syntax Tree (AST) nodes. The unnecessary lexical information within the Parse Tree is left out during its interpretation as syntactical elements.

c) From AST to Control Graph

The AST is then converted to basic blocks of Python bytecode instructions, which are emitted when each AST node is visited. Basic blocks are blocks of code that have a single entry but can have multiple exits. These blocks represent the Control Flow Graph (CFG), which indicates the code's possible execution paths.

d) From Control Graph to Code Object

Finally, the Control Flow Graph is flattened and its output, along with some data needed for execution, is used to generate Code Objects.

3. **Interpretation:** This stage executes the Code Object's bytecode in the virtual machine.

The **Python Virtual Machine** is a stack-based virtual machine that converts bytecode instructions to machine code. It uses three types of stacks (Bennett, 2018):

- **Call Stack:** This is the main structure of the Virtual Machine. The bottom of the stack is the entry point of the program. The stack contains frames, one for every presently active function call. When a function call is made, a frame is pushed to the stack and when the function returns, the frame is pooped off. For each frame, an Evaluation Stack and a Block Stack are created.
- **Evaluation Stack:** The execution of a function is made by interacting with this stack. Data is pushed, altered and popped of this stack accordingly.
- **Block Stack:** This stack is used to oversee control structures. It keeps track of which blocks (loops, try/except, continue, break, ...) are active at any given moment.

2.3.6 *MIPS Virtual Machine*

MIPS is a register-based virtual machine based on a Reduced Instruction Set Computer (RISC) architecture (see Figure 15). It was developed to increase performance with deep pipelines. It was largely used for embedded systems and video games consoles (Vaz, 2017). It is also used for educational purposes even though it is quite complex and difficult to learn.

The MIPS default emulator is the SPIM simulator, but there are several other intricate simulators designed to run this elaborate machine.

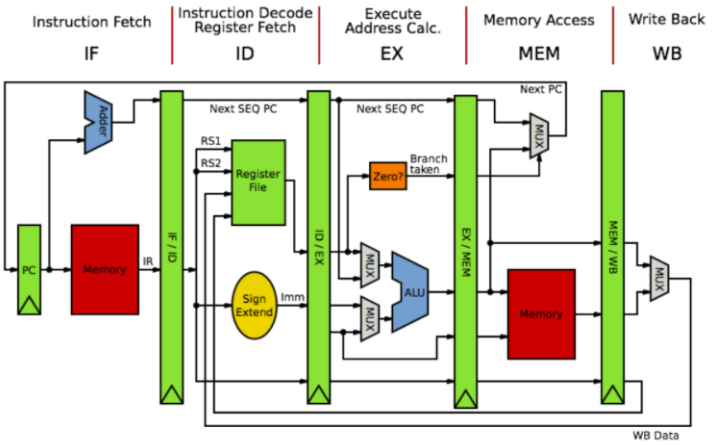


Figure 15: MIPS's Architecture.
Image taken from (Vaz, 2017).

2.3.7 *WebVm*

In University of Beira Interior, Nuno Gaspar and Simão Melo de Sousa (Gaspar and de Sousa, 2009) had stumbled upon the same problem, the need of a web-based tool to easily teach about compilers. As such, they created a web-based host platform for pedagogical virtual machines.

This web application in which the user could choose one of the implemented Virtual Machines, upload the code he wishes to run and either visualize the result or a step by step visualization of the machine's state evolution. Moreover, they provide the option to increment the number of virtual machines available, making the system more versatile although more complex.

In conclusion, the concept is the same: to facilitate the learning process by providing a step by step visualization of the machine's state. .

2.4 SUMMARY

Throughout this chapter, it was attempted to distinguish the various concepts of virtual machines, while also giving a bit of insight into the appearance and upsurge of virtual machines. Additionally, some examples of virtual machines were showcased and their features studied. Table 1 demonstrates some of their differences.

	Virtual Machines						
	IBM Cloud	Java	Google Colab	Parrot	Python	MIPS	WebVM
stack-based	x	x			x		x
register-based				x		x	x
online	x		x				x
cross-platform		x		x	x		x
simple			x		x		

Table 1: Virtual Machines - Feature Comparison.

The goal of this Master's Project is to create a Virtual Machine for educational purposes, to be studied and hopefully mastered during only one semester. To that end, it must have a very simple architecture and instruction set in order to allow for easy and fast learning. In addiction, it must preferably be in an online format to facilitate its accessibility.

Taking into account these intentions, none of these virtual machines meets the intended requirements. There was a previous attempt to use the Java Virtual Machine but it revealed itself too complex for such a short amount of time. Since WebVM was created with the same educational goal, it presents itself as a high contender to satisfy the selected requirements. However, this project aims for a more appealing solution in order to captivate students interest.

As such, this project is presented as an essential solution for this educational goal. EWVM focuses only on a low-level language and tries to maximize user experience in a simpler non intimidating way

3

VM, AN OVERVIEW

As stated earlier in this document, there is a Virtual Machine used in the Language Processing Course at Minho's University to teach students about compilers and code generation. This VM, a Stack-Machine implemented in C, was developed by Jean-Christophe Filliatre at LRI/Universit de Paris/Saclay and it runs programs in VM-Assembly language.

In addition to the Virtual Machine not being easily accessible, there are also some difficulties in executing it to its full capacity and features, notably the fact that not all computers are compatible with all needed software.

3.1 ARCHITECTURE

VM is composed of several structures that allow it to function correctly and in an organized manner, storing information in a systematically arranged configuration. As one can see in Figure 16, there are three main memory blocks, two of them being stacks, and two additional heaps. The two essential constituents that are needed to run the most basic programs are the **Operand Stack** and the **Code Zone**

The **Code Zone** is, as implied by the name, the memory block that holds the instructions uploaded by the programmer.

The **Operand Stack** is a pile that contains numbers (integers or reals) and addresses. It is where the instructions operate, therefore it is used to store all the operands needed o be processed by the program operators.

In order to be able to manipulate more complex data than numbers, there are two heaps, **Strings Heap** and **Structs Heap**, to which the addresses in the Operand Stack may point to. VM is also equipped with four **registers**, aimed at the management of the memory components and responsible for the proper functioning of the machine.

- **Program Counter (PC):** points to the current instruction in the code zone, i.e., the next to be fetched and executed.
- **Stack Pointer (SP):** points to the top of the operand stack, i.e., the first free cell.

- **Frame Pointer (FP):** points to the local variables base in the operand stack.
- **Global Variables Pointer (GP):** holds the global variables base address.

And finally, a more complex program with functions and local variables, justifies the need for the other component of this machine, the **Call Stack**. This pile contains pairs of pointers *(i, j)* that save present execution context (PC and FP) before a JUMP is executed to run the code of the called function. As such, every time a RETURN occurs, meaning that the function code has finished executing, the machine can recover the previous context and resume the normal execution. In the cell, the **Pointer *i*** holds the **PC** address and the **Pointer *f*** holds the **FP** address.

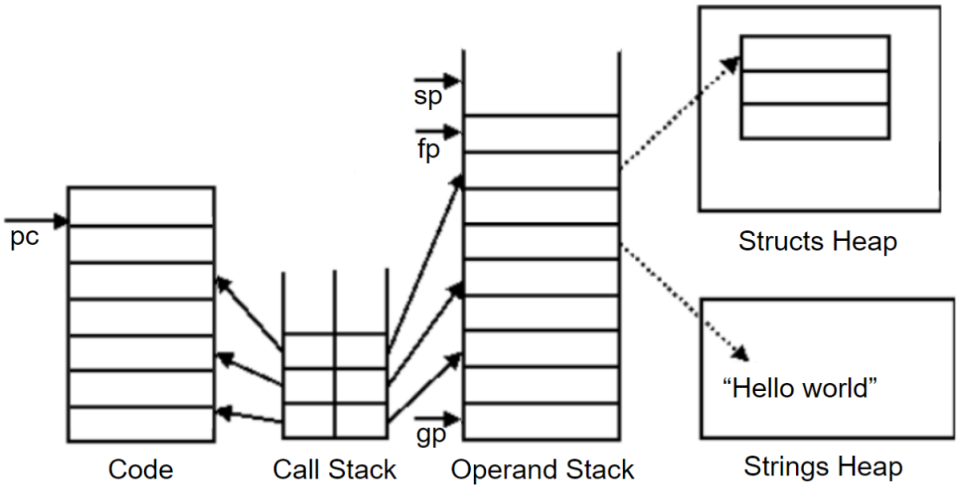


Figure 16: Current VM's Architecture.

3.2 FUNCTIONING PRINCIPLE

The Virtual Machine stores in its Code Zone memory code the sequence of instructions that compose the program (machine code) provided by the programmer. Consequently, it works by accessing that block of memory and going through it. Looking at each memory cell pointed by the PC register, the VM checks for an operator, iterates through its operands, if there are any, and executes the instruction. This process repeats until the end of the instruction sequence that might be identified by a specific operation, STOP, completing the program execution.

3.3 INSTRUCTIONS

Each machine instruction, which may be preceded by a tag (label) followed by a colon, is a a machine operation that may accept up to two parameters. The arguments can be integers, real numbers, chains of characters delimited by quotation marks (*strings*) or symbolic tags (labels) assigned to a code zone.

3.3.1 *Instruction Set*

This subsection lists all the instructions recognized and supported by VM grouped by their functionality.

The Instruction Set covers the following categories:

- **Basic Operations:** simple operations involving the various data types
- **Data Manipulation:** operations that alter the stack
- **Input-Output:** operations that engage the user
- **Control Operations:** operations that address jumps in the code's order
- **Start/Finish:** operations regarding the program's execution state

Basic Operations - Integers (all the operands handled must be integers):

ADD: takes *n* and *m* from the pile and stacks the result *m* + *n*
SUB: takes *n* and *m* from the pile and stacks the result *m* - *n*
MUL: takes *n* and *m* from the pile and stacks the result *m x n*
DIV: takes *n* and *m* from the pile and stacks the result *m / n*
MOD: takes *n* and *m* from the pile and stacks the result *m mod n*

NOT: takes *n* from the pile and stacks the result *n = 0*
INF: takes *n* and *m* from the pile and stacks the result *m* < *n*
INFEQ: takes *n* and *m* from the pile and stacks the result $m \leq n$
SUP: takes *n* and *m* from the pile and stacks the result *m* > *n*
SUPEQ: takes *n* and *m* from the pile and stacks the result $m \geq n$

Basic Operations - Floats (all the operands handled must be real numbers):

FADD: takes *n* and *m* from the pile and stacks the result *m* + *n*
FSUB: takes *n* and *m* from the pile and stacks the result *m* - *n*
FMUL: takes *n* and *m* from the pile and stacks the result *m x n*
FDIV: takes *n* and *m* from the pile and stacks the result *m / n*

FCOS: takes *n* from the pile and stacks the result *cos(n)*
FSIN: takes *n* from the pile and stacks the result *sin(n)*

FINF: takes *n* and *m* from the pile and stacks the result *m* < *n*
FINFEQ: takes *n* and *m* from the pile and stacks the result $m \leq n$
FSUP: takes *n* and *m* from the pile and stacks the result *m* > *n*
FSUPEQ: takes *n* and *m* from the pile and stacks the result $m \geq n$

Basic Operations - Addresses:

PADD: takes an Integer *n* and an address *a* from the pile and stacks the address *a* + *n*

Basic Operations - Strings (all the operands handled must be String Heap addresses):

CONCAT: takes *n* and *m*, from the pile and stacks the concatenated strings (*string ns* + *string ms*) address

Basic Operations - Structured Blocks Heap:

ALLOC - integer *n*: allocates a structured block, sized *n*, and stacks its address
ALLOCN: takes an integer *n* from the pile and allocates a structured block, sized *n*, and stacks its address
FREE: takes an address *a* from the pile and frees its allocated structured block

Basic Operations - Equality (the operands handled should have the same type):

EQUAL: takes *n* and *m* from the pile and stacks the result *n* = *m*

Basic Operations - Conversion:

ATOI: takes a String Heap address from the pile and stacks its string's conversion to an integer
(it fails if the string doesn't represent an integer)
ATOF: takes a String Heap address from the pile and stacks its string's conversion to a real number
(it fails if the string doesn't represent a real number)
ITOF: takes an integer from the pile and stacks its conversion to a real number
FTOI: takes a real number from the pile and stacks its conversion to a whole number - by removing its decimals
STRI: takes an integer from the pile, converts it to a string and stacks its address
STRF: takes a real number from the pile, converts it to a string and stacks its address

Data Manipulation - Move (load) to the pile:

PUSHI - integer *n*: stacks *n*
PUSHN - integer *n*: stacks *n* times the integer 0
PUSHF - real number *n*: stacks *n*
PUSHS - string *n*: archives *s* in the String Heap and stacks its address
PUSHG - integer *n*: stacks the value found in *gp[n]*
PUSHL - integer *n*: stacks the value found in *fp[n]*
PUSHSP: stacks the value of the register *sp*
PUSHFP: stacks the value of the register *fp*
PUSHGP: stacks the value of the register *gp*

LOAD - integer *n*: takes an address *a* from the pile and stacks the value found in *a[n]* in the pile or in the heap (depending on *a*)
LOADN: takes an integer *n* and an address *a* from the pile and stacks the value found in *a[n]* in the pile or in the heap (depending on *a*)
DUP - integer *n*: duplicates and stacks the *n* values of the top of the pile
DUPN: takes the integer *n* from the pile and duplicates and stacks the *n* values of the top of the pile

Data Manipulation - Take from the pile:

POP - integer *n*: takes *n* values from the pile
POPN: takes the integer *n* from the pile and takes *n* values *m* from the pile

Data Manipulation - Archive:

STOREL - integer *n*: takes a value from the pile and stores it in *fp[n]*
STOREG - integer *n*: takes a value from the pile and stores it in *gp[n]*
STORE - integer *n*: takes a value *v* and an address *a* and stores *v* in *a[n]* in the pile or the heap (depending on *a*)
STOREN: takes a value *v*, an integer *n* and an address *a* and stores *v* in *a[n]* in the pile or the heap (depending on *a*)

Data Manipulation - Miscellaneous:

CHECK - integers *n* and *p*: checks that at the top of the pile there's an integer *i* such that $n \leq i \leq p$ (*it throws an error if this is false*)
SWAP: takes the values *m* and *n* from the pile and stacks *m* followed by *n*

Input-Output:

WRITEI: takes an integer from the pile and prints its value
WRITEF: takes a real number from the pile and prints its value
WRITES: takes a String Heap address from the pile and prints its string
READ: reads a string from the keyboard, stores it in the String Heap and stacks its address

Control Operations - register *pc*:

PUSHA - *label*: stacks *label*'s code address
JUMP - *label*: assigns the *label*'s code address to the register *pc*
JZ - *label*: takes a value v from the pile and if:

- ***v* = *0***, assigns the *label*'s code address to the register *pc*
- ***v*** $\neq$ **0**, *increments register pc by* 1

Control Operations - procedures:

CALL: takes an label's address *a* from the pile, saves *pc* and *fp* in the Call Stack and assigns *a* to *pc* and the current *sp*'s value to *fp*.
RETURN: assigns the current *fp*'s value to *sp*, reinstates the values *fp* and *pc* from the Call Stack and increments *pc* by 1

Start and finish:

START: assigns *sp*'s value to *fp*
NOP: does not do anything
ERR - string *x*: throws an error with message *x*
STOP: stops program execution

3.4 SUMMARY

As the goal is to replicate and improve upon this machine, it was important to make a deep analysis of its architecture and functioning. This study allows for an accurate understanding of the VM's functionalities and instructions, serving as a reliable basis for the development of the new tool.

4

EWVM - PROPOSAL

The goal for creating this Educational Web Virtual Machine, EWVM, is to replicate the current VM Virtual Machine behaviour with a superior graphical user interface along with easy accessibility. To this end, the new Virtual Machine will be developed as a Web Application and will embody the same architecture and behaviour of the current one. As such, the new VM will contain the same memory components, **Code Zone, Call Stack, Operand Stack, String and Structured Blocks Heaps**, and its four registers, **Program Counter (PC), Stack Pointer (SP), Frame Pointer (FP)** and **Global Variables Pointer (GP)**. Evidently, the instruction set and instructions format will remain the same, apart from some additional features implemented to facilitate some basic operations that are missing in the present instruction set, such as the boolean operators AND and OR.

The Virtual Machine will be embedded in the *back-end* server of the Web Application, as depicted in Figure 17. The interface, in *front-end*, will take the programmer's code, send it to the server and wait for the results. In the server, the Program Code will pass through an **Assembler** which will translate it into Machine Code and send it into the **Run Engine**, where the Virtual Machine will be. The VM will then execute the code as explained and send back the result.

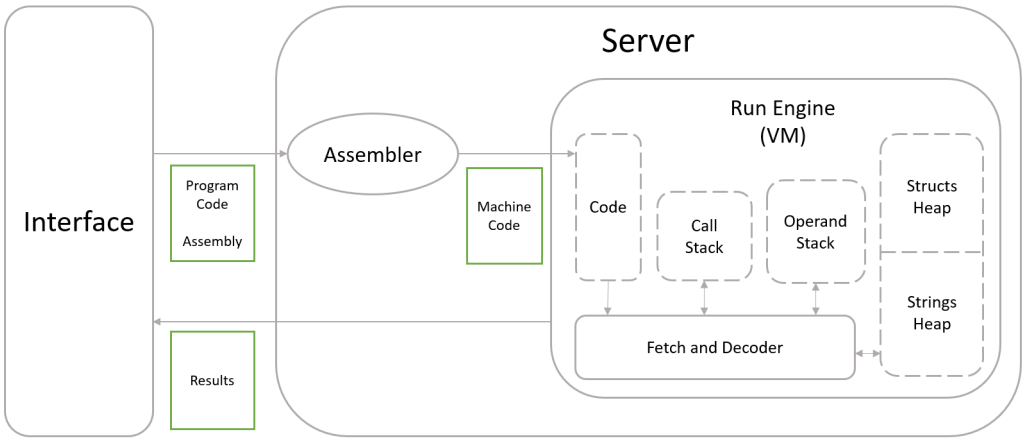


Figure 17: New VM's Architecture.

The interface will be composed of three main areas, aiming to offer more powerful features while remaining user friendly (see its mock-up in Figure 18). Each sector will have various features implemented for user interaction, as described below.

- **Code Sector:** contains a text area where the user can write the program and the following buttons and features:
  - **Clickable Instructions:** by clicking on a code instruction (operator mnemonic), a text box appears with the operation description.
  - **Upload File:** reads the selected file and uploads its content (an Assembly program), displaying it in the text area of this sector;
  - **Save to File:** downloads into an external file the code (Assembly program) in the text area;
  - **Run:** sends the assembly code in the text area to the Back-End to be assemblied and executed;
- **Animation Sector:** holds a container in which the user can visualize the virtual machine's internal state in each step of the code execution. It also contains the following features:
  - **Numbering:** each step displayed is numbered and sorted by execution order
  - **Navigate:** by clicking on arrows, the user can move forward or backwards through the steps; it is also possible to move directly to the first or last step
- **Interaction Sector:** composed of two windows, one where the machine writes its outputs and the other from where it reads the user's inputs

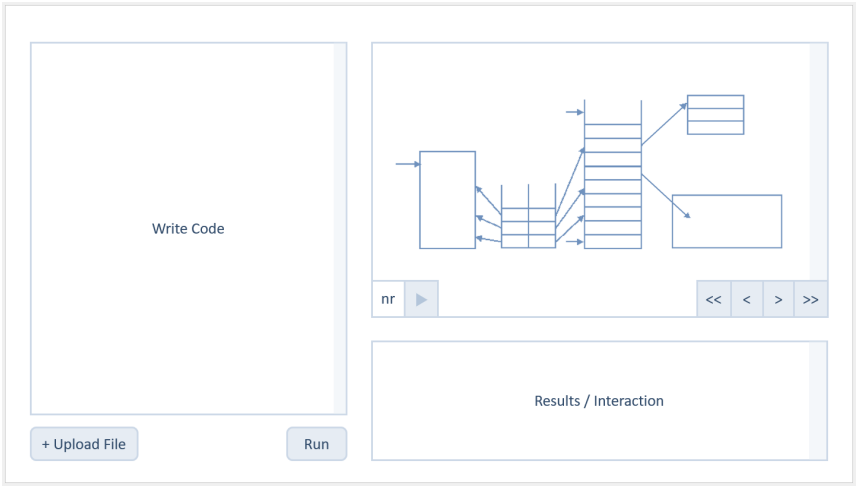


Figure 18: Website's Mockup.

5

EWVM - THE ASSEMBLER

The Web Application is written in JavaScript. It was developed in **Node.js**[1], a back-end event-driven JavaScript runtime environment that executes JavaScript code outside the web browser. It is designed to build scalable network applications and supports the **Express**[2] framework, which offers a set of features for web applications.

Accordingly, since the program is written in JavaScript, the **Assembler** has been developed in **peggy**, a JavaScript API. It is a parser generator that integrates both lexical and syntactic analysis and is based on a context free grammar formalism. The input grammar is written in Extended BNF (Backus Naur Form). Each grammar rule can be associated to a semantic action that is a fragment of javascript code written between curly brackets. Peggy processes the input grammar and generates a fast and powerful compiler which receives an input text and returns either the results or a thorough and clear error report.

The grammar written to generate the Assembler, presented in Listing 1, was written using Peggy online version[3]. It was created to analyse the assembly code written by the user and check if it is lexically and syntactically correct according to the Instruction Set rules. The grammar also semantically analyses the input and either detects an error or translates the received instructions to Machine Code.

```
1  Code = Line* _
2
3  Line = (_ Instruction)  ([ \t\r]* Comment)*
4      / Comment
5
6  Instruction = Label ':'
7      / Inst_Atom
8      / Inst_Int _ Integer
9      / "pushf" _ Float
10     / "pushs" _ String
```

1 https://nodejs.org/en/
2 https://expressjs.com/
3 https://peggyjs.org/

```
    / "err" _ String
    / "check" _ (Integer _ "," _ Integer)
    / "jump" _ Label
    / "jz" _ Label
    / "pusha" _ Label

Inst_Atom = "stop" / "start" / "add" / "sub" / "mul" / "div" / "mod" / "not" / "infeq"
          / "inf" / "supeq" / "sup" / "fadd" / "fsub" / "fmul" / "fdiv" / "fcos" / "fsin"
          / "finfeq" / "finf"  / "fsupeq" / "fsup"  / "concat" / "equal" / "atoi" / "atof"
          / "itof" / "ftoi" / "stri" / "strf" / "pushsp" / "pushfp" / "pushgp" / "loadn"
          / "storen" / "swap" / "writei" / "writef" / "writes" / "read" / "call"
          / "return" / "allocn" / "free" / "dupn" / "popn" / "padd" / "nop"

Inst_Int = "pushi" / "pushn" / "pushg" / "pushl" / "load" / "dup" / "pop"
         / "storel" / "storeg" / "store" / "alloc"

Comment = "//" [^\n]*

Label "label" = [a-zA-Z0-9]+

String = '"' Content*  '"'

Content = "\\n" / [^"]

Integer "integer" = ("+"/"-")? _ [0-9]+

Float = ("+"/"-")? _ Integer(.Integer)?

_ "whitespace" = Space*

Space = [\n] / [ \t\r]
```

Listing 1: Grammar

To transform the given Program Code into Machine Code, the generated compiler parses the Program Code and replaces the instruction mnemonics with their respective internal codes while managing the labels.

The Machine Code is saved in the form of an array and in each index an Instruction Code is kept.

```
Machine Code:
[ Instruction Code 1, Instruction Code 2, ...]
```

Each Instruction Code comprises the instruction internal code and its operands. To enable future connection between the code area and the animation, the line number in which the instruction was written on is also saved.

```
Instruction Code:
[ line number, instruction internal code, operands ]
```

At the end, the Assembler replaces the label addresses in the array with their respective array index.

As stated, a Machine Code is assigned to each instruction. The referred assignments are made according to the following table:

```
Machine Code - Instruction
0 - stop        15 - fdiv       30 - pushsp     45 - popn       60 - err
1 - start       16 - fcos       31 - pushfp     46 - padd       61 - check
2 - add         17 - fsin       32 - pushgp     47 - pushi      62 - jump
3 - sub         18 - finf       33 - loadn      48 - pushn      63 - jz
4 - mul         19 - finfeq     34 - storen     49 - pushg      64 - pusha
5 - div         20 - fsup       35 - swap       50 - pushl      65 - nop
6 - mod         21 - fsupeq     36 - writei     51 - load
7 - not         22 - concat     37 - writef     52 - dup
8 - inf         23 - equal      38 - writes     53 - pop
9 - infeq       24 - atoi       39 - read       54 - storel
10 - sup        25 - atof       40 - call       55 - storeg
11 - supeq      26 - itof       41 - return     56 - store
12 - fadd       27 - ftoi       42 - allocn     57 - alloc
13 - fsub       28 - stri       43 - free       58 - pushf
14 - fmul       29 - strf       44 - dupn       59 - pushs
```

This simple and effective Assembly Grammar, that defines the low-level programming language used to program the VM, is used as input to a Compiler Generator to build automatically the Assembler (see Figure 19). The Assembler reads the programmer's code to be executed by the Virtual Machine, provided through the Interface and translates it to be executed by the Run Engine.

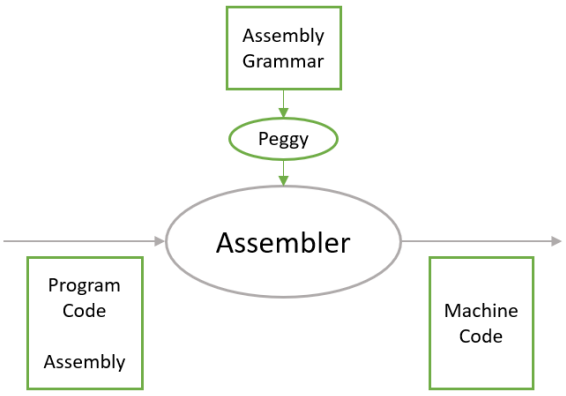


Figure 19: Assembler.

In Figure 20 it is displayed an example of the Web-based Compilers Generator system Peggy, used to develop the VM Assembler.

Figure 20: Assembler development using Peggy Generator.

In this tool, the Assembler generated by peggy is called each time the user clicks in the Run button, receiving as an input the text introduced by the programmer at the Code Sector and sending its result, the machine code generated, to the Virtual Machine.

6

EWVM - THE VIRTUAL MACHINE

The Virtual Machine iterates through the Machine Code generated by the Assembler. For each instruction the virtual machine runs the corresponding procedure, accessing memory cells (to read or write data), executing arithmetic or logic operations and altering the memory blocks according to the computed results.

6.1 MEMORY BLOCKS

CODE STACK represents the memory sector of a real machine where the program is uploaded and is implemented through an array where the Machine Code is stored.

OPERAND STACK represents the memory sector or registers bank of a real machine. It is used to store temporary data along the computation of an expression or the execution of another operation that requires intermediate data storage. It is implemented through an array of either numeric values or references to other memory blocks. Each reference is stored in the following way:

```
Code Stack Reference:          code#index
                               example: code#0

Operand Stack Reference:       stack#index
                               example: stack#3

Struct Heap Reference:         struct#heap_index#struct_index
                               example: struct#1#0

String Heap Reference:         string#index
                               example: string#0
```

CALL STACK corresponds in a real machine to a bank of registers for temporary storage of control data to return from subprograms. It is implemented through an array where each element carries two numeric values. The first one corresponds to an index of the Code Stack that stores the previous value of PC (program counter) register to remember the next instruction to be executed after returning from a function call. The second one corresponds to an index of the Operand Stack that stores the previous value of FP (frame pointer) register to remember the local memory base-address to recover local data memory after returning from a function call.

HEAPS

- The String Heap is an array that stores sequentially all the Strings handled by the program along its execution (both literal strings inserted in the source program and strings read during the program execution).
- The Struct Heap is an array of Arrays. Each array represents a block of allocated memory created by the program.

6.2 FLOW CHART EXAMPLE

This section is going to demonstrate the Virtual Machine's behaviour when running a program. It will use a simple program as an example and explain the steps and transformations it goes through until its execution is completed and the result is sent to back to the user.

The program chosen as the example combines the following instructions:

1. **PUSHI 5:** stacks 5
2. **PUSHI 6:** stacks 6
3. **SWAP:** takes the values 6 and 5 from the pile and stacks 6 followed by 5

The program written by the user is sent to the Server. In the server, it will pass through the Assembler before going to the Virtual Machine. As previously explained, this step translates the Program Code into Machine Code, a format ready to be processed by the Virtual Machine.

The program is transformed as followed:

1. **PUSHI 5** $\longrightarrow$ [1, 47, 5]
    - line_number: 1
    - instruction_internal_code: 47
    - operands: 5
2. **PUSHI 6** $\longrightarrow$ [2, 47, 6]
    - line_number: 2
    - instruction_internal_code: 47
    - operands: 6
3. **SWAP** $\longrightarrow$ [3, 35]
    - line_number: 3
    - instruction_internal_code: 35

These Instruction Codes form the Machine Code which will be received by the Virtual Machine:

```
Machine Code = [ [1, 47, 5], [2, 47, 6], [3, 35] ]
```

The execution of the code in the Virtual Machine runs the following algorithm:

```
foreach (line_number, instruction_internal_code, operands) in machine_code:
    execute(instruction_internal_code, operands)
    save_state(line_number)
```

The Virtual Machine iterates through the Machine Code instructions and executes them. After every instruction execution, the machine's state is saved. In the end, these states are sent to GUI in order to present the user with the step-by-step machine state display.

```
State - array of all the machine states during the program's execution:
[ Machine State 1, Machine State 2, ... ]

Machine State - machine state after an executed instruction:
[ line number, operand stack, call stack, string heap, struct heap ]
```

The functioning and steps of this algorithm is illustrated in Figure 21.

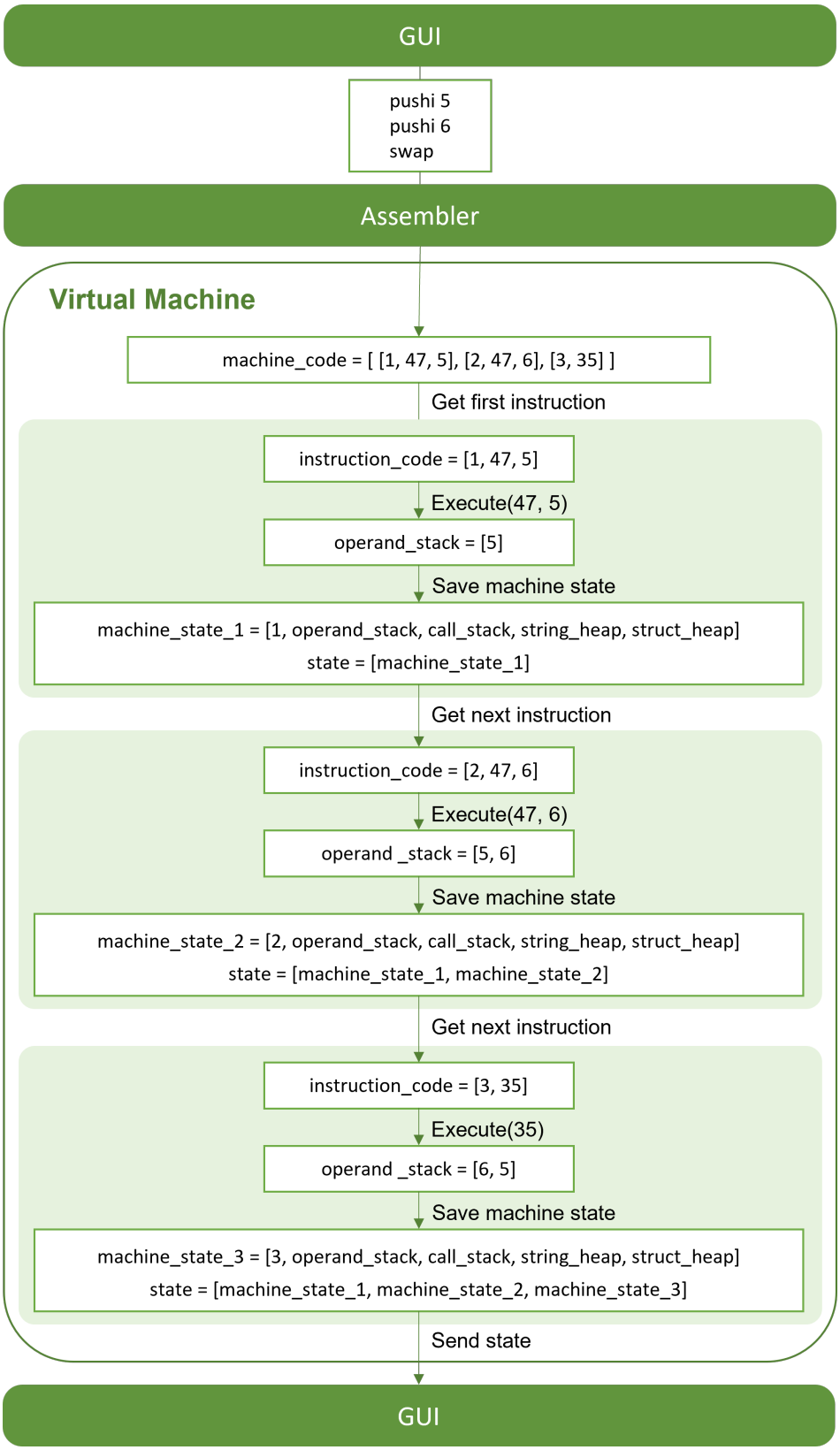


Figure 21: EWVM Flow Chart example

7

EWVM - THE GRAPHICAL USER INTERFACE

Besides the features proposed in Section 4, other mechanisms were implemented in the various sectors (windows) of the website (shown in Figure 22) in order to improve this machine as an educational tool.

It is called *machine state* to the set of values stored in the data memory blocks (the two stacks and the two heaps) together with the values contained in the four registers after the execution of one specific instruction of the program stored in the code memory. Each sector enables the user to choose a machine state he wants to visit. After the choice of a state in one sector, the values displayed in the other sectors also change accordingly.

- **Code Sector (the leftmost window in the screenshot of Figure 22):**
  - **State Choice:** after a complete run of the input program, the user can, by clicking on an instruction line number, select the state subsequent to that instruction's execution; if the instruction has been executed more than once, additional clicking should iterate through the respective states.
  - **Connection:** the code sector highlights the instruction line number corresponding to the displayed state
- **Animation Sector (the central, main, window in the screenshot of Figure 22):**
  - **State Choice:** this sector contains a group of four navigation buttons that allow the user to visualize the various machine states
  - **Connection:** illustrates the selected machine state
- **Interaction Sector (the bottom window, at the center, in the screenshot of Figure 22):**
  - **State Choice:** by clicking on the text visible on the output window, it is possible to select the state in which that text was printed
  - **Connection:** points out which text has already been printed (or is being printed) according to the machine state

The developed GUI (Graphical User Interface) can be observed in Figure 22. Having all the discussed features and connections, it achieves the desired educational version of the virtual machine VM, EWVM.

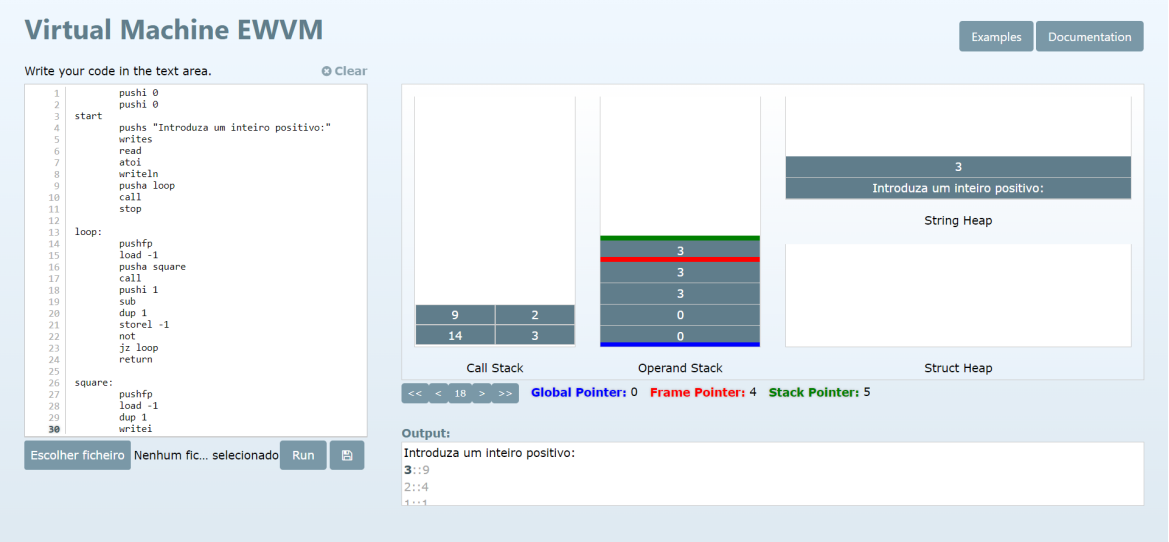


Figure 22: A screenshot of EWVM interface

As planned, this GUI allows the users to analyse the state of the machine in each step of the code execution. The user can observe the values of registers and memory blocks using the information displayed in the Animation Sector, while the corresponding instruction is highlighted in the Code Sector. Meanwhile, in the Interaction Sector, the printed text is differentiated by colour.

These features enable the user to get a clear understanding of the actual effect of each instruction on the components of the machine (this is the so-called *operational semantics* of the input program).

In this manner, the platform works to decrease the learning curve, allowing students to have a better understanding of the machine's behaviour by maximizing the amount of presented information.

As it is essential to the learning experience, a users-guide was added where instructions are listed and explained. This **Manual** containing the code's Documentation can be observed in Figure 23.

Figure 23: A screenshot of the EWVM Users' Manual

In order to give users a little more guidance and help, the website offers **Program Examples** (see Figure 24) categorized in terms of topic and difficulty. These are easily accessed and ran, providing the user the opportunity to explore these programs and learn from them.

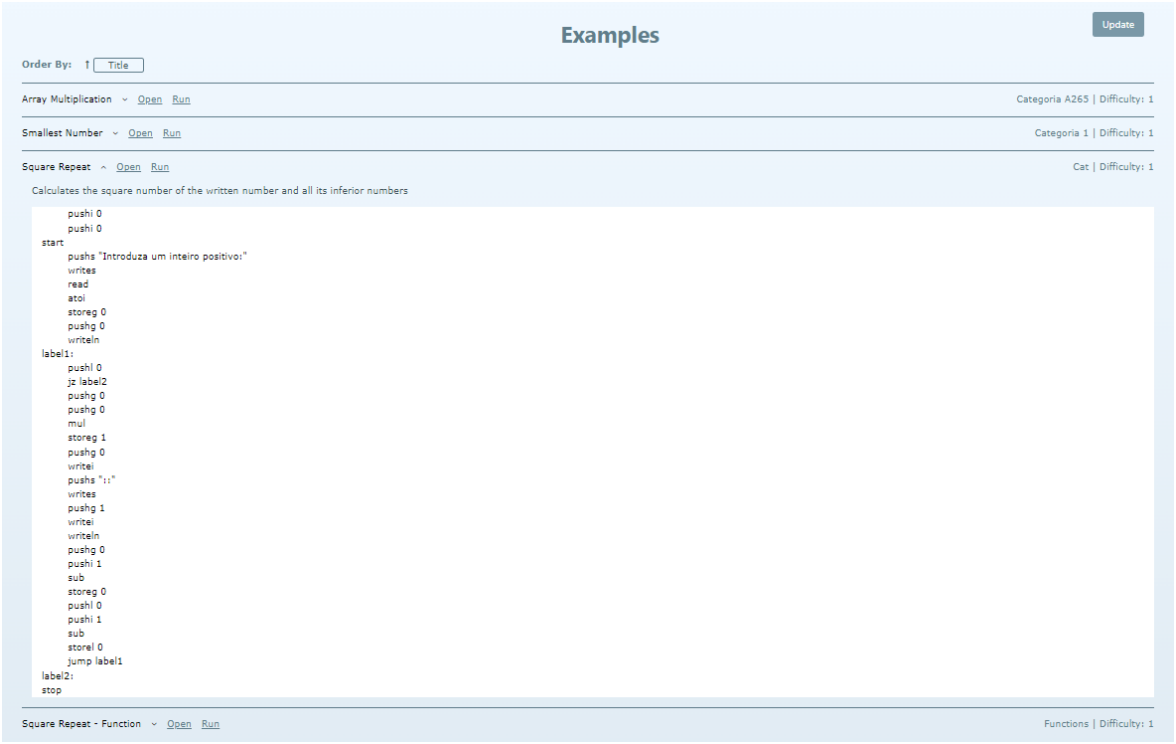


Figure 24: EWVM - Examples

Each example's code is kept in its own file saved in a specific folder. To keep record of the examples, a **JSON file** was created containing a list of each example's metadata, including the filename of the example's code. This JSON file is written according to the following json-schema:

```
{
    "$schema": "https://json-schema.org/draft/2022-06/schema",
    "$id": "https://example.com/examples-schema.json",
    "title": "Educational Code Examples",
    "type": "object",
    "required": [ "examples"],
    "properties": {
      "examples": {
        "items": {
          "type": "object",
          "required": [ "title", "category", "file" ],
          "properties": {
              "title": {
                "description": "Unique identifier/title of an example",
                "type": "string"
              },
              "category": {
                "description": "Category of the example",
                "type": "string"
              },
              "description": {
                "description": "Description of the example",
                "type": "string"
              },
              "difficulty": {
                "description": "Difficulty of the product.",
                "type": "integer"
              },
              "file": {
                "description": "Filename of the example's code",
                "type": "string"
  } } } } }   }
```

7.1 DEMONSTRATION

This section aims to clarify how the sectors are connected by using a simple program. This programs asks the user for two input numbers and outputs their sum. In Figure 25 the program has been run and is waiting for input.

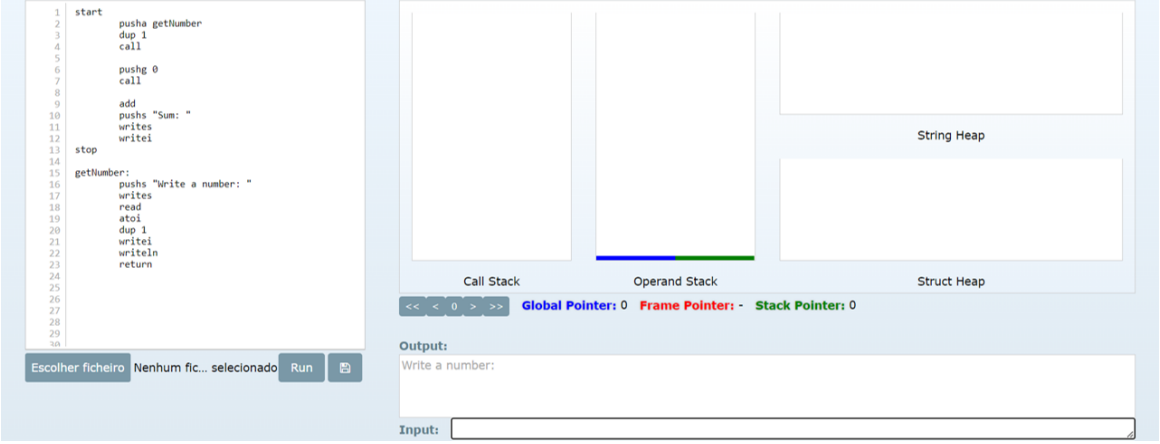


Figure 25: Demonstration - Program

In Figure 26 it can be seen what happens when the user clicks on the Go-to-last Button. In this case, the Animation goes to the instruction that asks the user for input. In the Code Sector the line number of its instruction is highlighted. In the Interaction Sector the output that has already been printed is blackened.

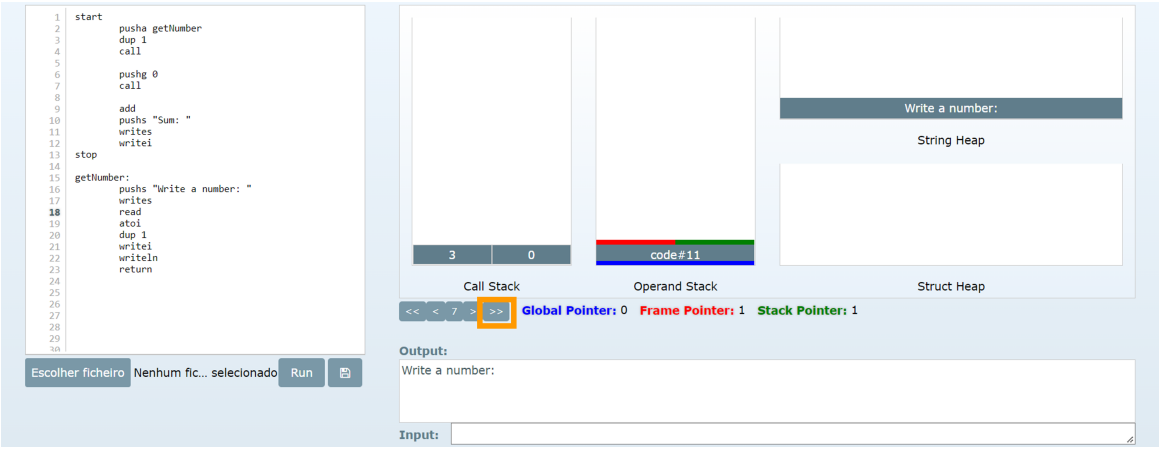


Figure 26: Demonstration - Go-to-last Button connection

In Figure 27 it can be seen what happens when the user clicks on the Go-to-next Button. The Animation moves to the next step and, as seen before, the line number is highlighted and the output that has not been printed remains grey.

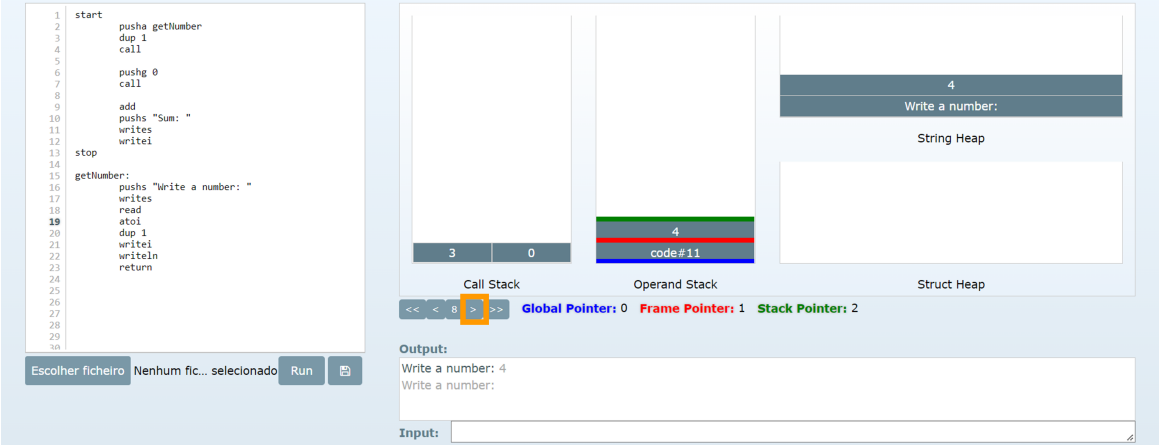


Figure 27: Demonstration - Go-to-next Button connection

In Figure 28 it can be seen what happens when the user clicks on a line number. The number is highlighted, the Animation Sector displays the corresponding step and, in this case, the Output highlights the string that the chosen instruction is printing. As demonstrated in the figure, if the program passes through the line number instruction more than once, clicking on it again will iterate through the states.

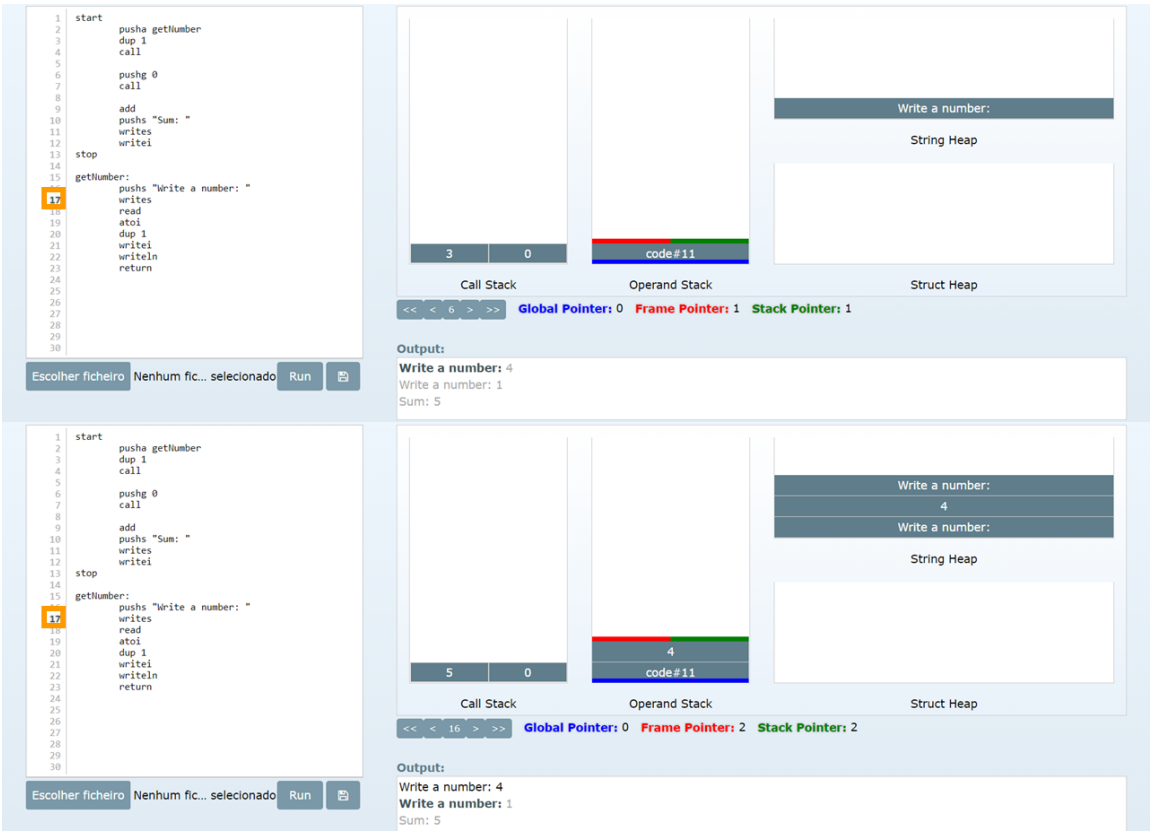


Figure 28: Demonstration - Line number connection

In Figure 29 it can be seen what happens when the user clicks on the output. Its string and the line number of the instruction which prints it are highlighted and the Animation Sector displays the corresponding state.

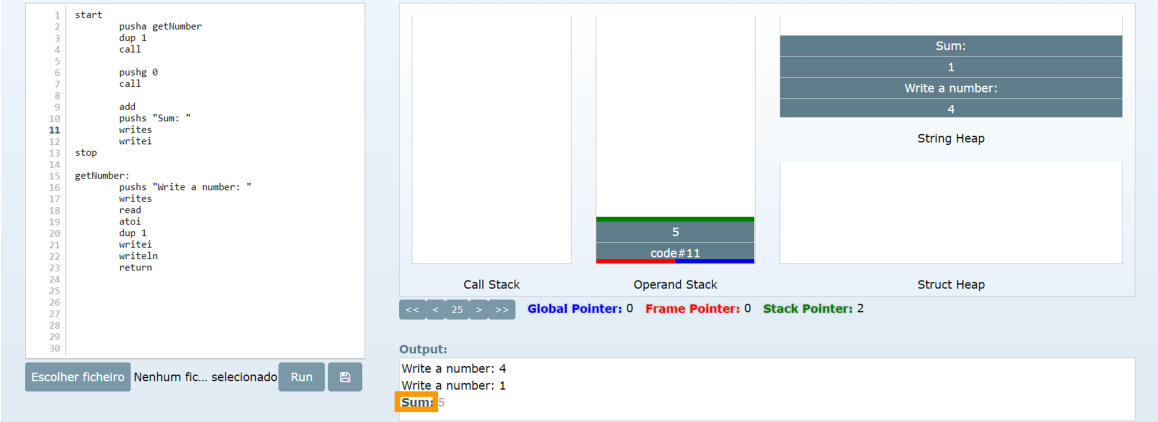


Figure 29: Demonstration - Output connection

Making all these connections ensures the user a better understanding of how each instruction works, since it is clear the effect each one has on the machine state and program execution.

8

CONCLUSION

This document dives into the subject of virtual machines, beginning with an explanation on the topic and an insight into what they are used for. It then clarifies the motivation for the Master's Project here described and points out the necessary steps to develop it and the process to achieve them.

A more in-depth look is then made into the world of Virtual Machines. At first, it plunges into its history and how it emerged, featuring the two types of virtual machines: Type I and Type II. It then proceeds to compare Stack-based Virtual Machines to Register-based Virtual Machines, explaining their functioning and weighing their virtues. In addition, it touches on the various categories of virtualization. Lastly, there is a study of a few current commonly used virtual machines, highlighting their features and architectures. It concludes that none of the specified virtual machines is suited to address the shortcomings this project aims to overcome.

Additionally, there is an overview of the existing virtual machine VM which includes an in-depth and elaborate description of its architecture, as well as an explanation of its functioning and a presentation and description of its instruction set.

A detailed characterization of the product this project hopes to develop was also included. The proposed Web Application structure is presented, explaining the interface-server connection and the server's components, Assembler and Run Engine. The latter represents the Virtual Machine, which has been designed based on the previous VM's architecture. A mock-up of the website is depicted, alongside a list of features considered for implementation.

Then the document dives into the implementation of the project which encompasses three components: Assembler, Virtual Machine and GUI.

The development of the Assembler is described and explained, revealing the technologies chosen to develop the project. The Assembly Grammar created is listed, along with a demonstration and explanation of its functioning and the part it plays in the code's execution.

Since the Virtual Machine's architecture and behavior were previously analysed and explained in the document, at this point the focus was on the more concrete implementation of its elements and values.

Finally, the Graphical User Interface is addressed. Having its features been previously listed, the document explores the connections and mechanisms of its system and displays the final product. In addition, the document discusses the Manual and the Project Examples created in order to provide a better educational platform, also explaining how the data is saved in the system.

All the goals set for this project have been met and the application can be considered a proper educational tool having achieved an optimal interaction between the user and connected sectors through a step-by-step approach.

At the moment, EWVM is available in a public URL: https://ewvm.epl.di.uminho.pt/. It was used in the Language Processing Course by more than 160 students and received very positive feedback. Even though EWVM is a Web Application, it has been dockerized to ease its installation, reducing the process to a command line execution.

Concerning future work, there is an intent to tune the VM language. For instance, the VM instruction set has some graphical instructions that, due to many difficulties in the graphical output, have not been explored in the present tool. The plan is to implement these instructions adding a panel to the interface with an HTML canvas or having the VM compiler produce SVG or other formats that browsers can display and animate.

BIBLIOGRAPHY

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers Principles, Techniques and Tools*. Addison-Wesley, 1986.

Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, Principles, Techniques, and Tools*. Pearson Education, Inc, second edition, 2007.

Sameer M AlNajdi, Malek Q Alrashidi, and Khalid S Almohamadi. The effectiveness of using augmented reality (ar) on assembling and exploring educational mobile robot in pedagogical virtual machine (pvm). *Interactive Learning Environments*, 28(8):964–990, 2020.

Steve Baca. Virtualization for newbies: Five types of virtualization, Nov 2021. URL https://www.globalknowledge.com/us-en/resources/resource-library/articles/virtualization-for-newbies-five-types-of-virtualization/. Global Knowledge.

James Bennett. An introduction to python bytecode, April 2018. URL https://opensource.com/article/18/4/introduction-python-bytecode.

J. P. Buzen and U. O. Gagliardi. The evolution of virtual machine architecture. In *Proceedings of the June 4-8, 1973, National Computer Conference and Exposition*, AFIPS '73, page 291–299, New York, NY, USA, 1973. Association for Computing Machinery. ISBN 9781450379168. doi: 10.1145/1499586.1499667. URL https://doi.org/10.1145/1499586.1499667.

Tom Chothia and Chris Novakovic. An offline capture the flag-style virtual machine and an assessment of its value for cybersecurity education. In *2015 USENIX Summit on Gaming, Games, and Gamification in Security Education (3GSE 15)*, Washington, D.C., August 2015. USENIX Association. URL https://www.usenix.org/conference/3gse15/summit-program/presentation/chothia.

Ayan Das. Advanced python: Bytecodes and the python virutal machine (vm) - part i, Jan 2019. URL https://medium.com/@dasayan05/advanced-python-bytecodes-and-the-python-virutal-machine-vm-part-i-b58ed526f2b.

Brian Davis, Andrew Beatty, Kevin Casey, David Gregg, and John Waldron. The case for virtual register machines. In *Proceedings of the 2003 workshop on Interpreters, virtual machines and emulators*, pages 41–49, 2003.

Simão Melo de Sousa. *Máquina Virtual para o projecto da disciplina de compiladores*. Dep. de Informática, Universidade da Beira Interior, Covilhão, Portugal, Set 2006.

Joseph A. Driscoll, Ralph M. Butler, and Joelle M. Key. A virtual machine environment for teaching the development of system software. In *Proceedings of the 42nd Annual Southeast Regional Conference*, ACM-SE 42, page 440–441, New York, NY, USA, 2004. Association for Computing Machinery. ISBN 1581138709. doi: 10.1145/986537.986647. URL https://doi.org/10.1145/986537.986647.

IBM Cloud Education. Private cloud, Apr 2020. URL https://www.ibm.com/cloud/learn/introduction-to-private-cloud.

Fabian Fagerholm. Perl 6 and the parrot virtual machine, 2005.

Nuno Gaspar and Simão Melo de Sousa. WebVm - A web-based host platform for pedagogical virtual machines. *Special issue of the journal Informática na Educação: teoria & prática*, 1 (4), jan/jun 2009. ISSN 1516-084X.

Xuelian Hu and Dong Han. The design, implementation and application of minijava/ad as an object-oriented compiler teaching model. In *2009 4th International Conference on Computer Science Education*, pages 1488–1491, 2009. doi: 10.1109/ICCSE.2009.5228571.

IBM Cloud Education. Virtual machines. https://www.ibm.com/cloud/learn/virtual-machines, Jun 2019. Accessed: 05-10-2021.

Obi Ike-Nwosu. Read inside the python virtual machine. URL https://leanpub.com/insidethepythonvirtualmachine/read#leanpub-auto-the-interpreter-state.

KAUSHIK K. Internal working of python, Aug 2021. URL https://medium.com/@kaushik.k/internal-working-of-python-415572929e7a.

Kexugit. Why have a stack?, Nov 2011. URL https://docs.microsoft.com/pt-pt/archive/blogs/ericlippert/why-have-a-stack.

Eric Kohlbrenner, Dana Morris, and Brett Morris. Virtual machine. URL http://denninginstitute.com/itcore/virtualmachine/ibm.htm.

Yunfa Li, Wanqing Li, and Congfeng Jiang. A survey of virtual machine system: Current technology and future trends. In *2010 Third International Symposium on Electronic Commerce and Security*, pages 332–336. IEEE, 2010.

Siben Nayak. Jvm tutorial - java virtual machine architecture explained for beginners, Jan 2021. URL https://www.freecodecamp.org/news/jvm-tutorial-java-virtual-machine-architecture-explained-for-beginners/.

Anh M Nguyen, Nabil Schear, HeeDong Jung, Apeksha Godiyal, Samuel T King, and Hai D Nguyen. Mavmm: Lightweight and purpose built vmm for malware analysis. In *2009 Annual Computer Security Applications Conference*, pages 441–450. IEEE, 2009.

Michael Prantl. Python internals: An introduction, Oct 2020. URL https://blog.sourcerer.io/python-internals-an-introduction-d14f9f70e583.

Yunhe Shi, Kevin Casey, M. Anton Ertl, and David Gregg. Virtual machine showdown: Stack versus registers. *ACM Trans. Archit. Code Optim.*, 4(4), January 2008. ISSN 1544-3566. doi: 10.1145/1328195.1328197. URL https://doi.org/10.1145/1328195.1328197.

Simplilearn. What is microsoft azure and how does it work: Simplilearn, Dec 2021. URL https://www.simplilearn.com/tutorials/azure-tutorial/what-is-azure.

Mark Vinod Sinnathamby. Stack based vs register based virtual machine architecture, and the dalvik vm, Sep 2012. URL https://www.codeproject.com/Articles/461052/Stack-Based-vs-Register-Based-Virtual-Machine-Arch.

Hasitha Subhashana. Understanding how java virtual machine (jvm) works, May 2021. URL https://hasithas.medium.com/understanding-how-java-virtual-machine-jvm-works-a1b07c0c399a.

TechVidvan. Jvm - java virtual machine working and architecture, Jun 2021. URL https://techvidvan.com/tutorials/java-virtual-machine/.

Damien Silva Vaz. Implementing an integrated syntax directed editor for liss. January 2017.

Jatinder Virdee. Virtualization on azure, Jan 2015. URL https://blog.sysfore.com/virtualization-on-azure/.