EWVM, a Web Virtual Machine to Support Code Generation in Compiler Courses

Sofia Teixeira[1]
Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

José Carlos Ramalho
Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Pedro Rangel Henriques
Centro ALGORITMI, Departamento de Informática, University of Minho, Braga, Portugal

Abstract

This paper describes a project which goal is to analyze and model a complete Virtual stack Machine (VM) environment and build a Web application with a graphical interface to deploy an environment to compile and execute VM programs. The new tool offers two main features: assembles and reports errors in programs written in the assembly language of the Virtual Machine; and animates the execution of the compiled code, displaying the internal state of the VM and providing an interface to control the execution step-by-step. In the paper, after discussing related concepts and works, a proposal to build such a tool, so far called EWVM, will be presented along the architecture drawn. A prototype will be shown, and its impact as an educational tool is argued.

2012 ACM Subject Classification Software and its engineering → Compilers; Software and its engineering → Virtual machines

Keywords and phrases Virtual Machine, Stack Machine, Assembler, Debugger, Compiler, Code Generation

Digital Object Identifier 10.4230/OASIcs.SLATE.2022.7

Supplementary Material *Software (Web Application)*: `https://ewvm.epl.di.uminho.pt/`

Funding This work has been supported by FCT – Fundação para a Ciência e Tecnologia within the R&D Units Project Scope: UIDB/00319/2020.

1 Introduction

A Virtual Machine (VM) is a software layer over the actual 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 [10].

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 [12]. The most obvious use for VMs is to run incompatible software or to test new operating systems. Some VMs, like Java Virtual Machine [16], were created to guarantee software compatibility, allowing different equipment or operating systems to have the same compiler. That feature is responsible for the well-known portability of Java applications, and it is also present in other popular languages as Perl or Python that use their own virtual machines, Parrot [7] and Python Virtual Machine [13].

[1] corresponding author


© Sofia Teixeira, José Carlos Ramalho, and Pedro Rangel Henriques;
licensed under Creative Commons License CC-BY 4.0

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 [2], such as teaching in different domains, namely cybersecurity [3], system software development [6] and others [9].

At the moment, a Virtual Machine, named VM, developed by Jean-Christophe Filliatre at LRI/Université de Paris Saclay for his Compiler courses [5], is being used at our University in the context of Language Processing courses to teach students about compilers and code generation. As a Learning Resource, VM is valuable due to its logic and clear working principle and to the simplicity of its reduced instruction set. However VM runtime environment, that interpreters the Assembly programs and allows their execution step-by-step for debugging purposes, is a stand alone application that is not easy to install and work with. This fact complicates its usage during the Compiler course and motivates us to start a new project aimed at developing a new, more user-friendly resource. As such, VM is the foundation of the new Web-based Virtual Machine that is introduced and explained along this paper.

This paper is organized in five sections. This section presents the project context, motivation and paper focus. Section 2 introduces the two main types of virtual machines and discusses their features. Moreover, in Section 2 there is also a description of the basic ideas underlying the VM in use at moment. Section 3 presents the proposed architecture to develop the new system, EWVM, that extends VM and makes it available in a Web browser. Section 4 defines the Assembly language recognized by EWVM and presents the prototype already available for tests and improvements. Section 5 closes the paper with some conclusions and future work.

2 Virtual Machines and related work

Virtual Machines bring 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.

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 to or 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 [15].

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 [15]. However that more efficient approach requires from the compiler an extra task that is complex: the registers management.

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 [14] 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 [4]. 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 [11], 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 [1].

2.1 Current Virtual Machine

As stated, a Virtual Machine is currently being used in the Language Processing Course at our university to teach students about compilers and code generation. The so-called VM is a Stack-Machine implemented in C that 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.

2.1.1 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 1, there are three main memory blocks, two of them being stacks, and two additional heaps in the machine. 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 main 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.

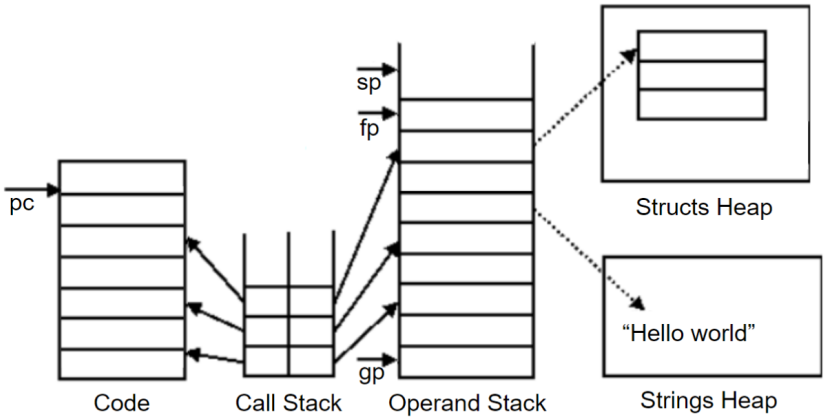

Figure 1 VM's Architecture.

2.1.2 Functioning Principle

The Virtual Machine stores in its Code Zone memory 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.

2.1.3 Instruction Set

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.

2.2 Related Work

In University of Beira Interior, Nuno Gaspar and Simão Melo de Sousa [8] had stumbled upon the same problem, the need of a web-based tool to easily teach about compilers. As such, they created a 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. However, EWVM focuses only on a low-level language and tries to maximize user experience in a simpler non intimidating way.

3 EWVM, proposal and architecture

The goal for creating this Educational Web Virtual Machine, EWVM, is to replicate the current Virtual Machine's 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 2. 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 turn 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.

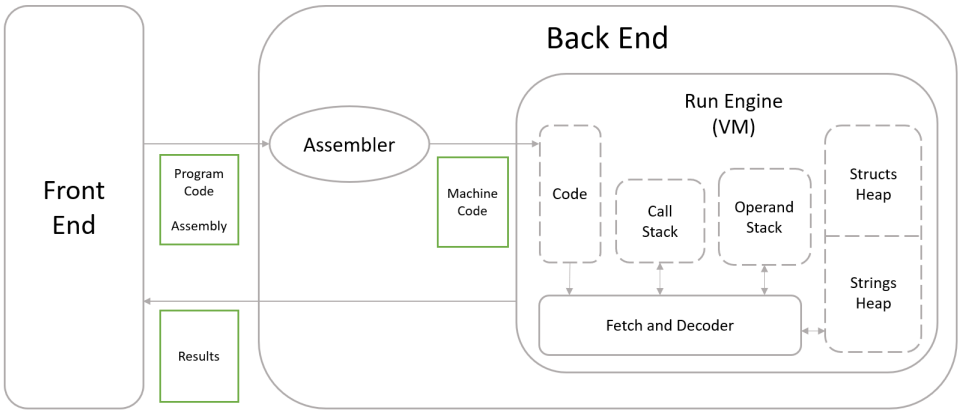

Figure 2 EWVM Architecture.

The interface will be composed of three main areas, aiming to offer more powerful features while remaining user friendly (see the final result in Figure 3). 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:
  - **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 display is numbered and sorted by execution order
  - **Navigate:** by clicking on arrows, the user can move forward or backwards through the displays; it is also possible to move directly to the first or last display
- **Interaction Sector:** composed of two windows, one where the machine writes its outputs and the other from where it reads the user's inputs

4 EWVM, the tool

The Web Application is written in JavaScript. It is being developed in **Node.js**[2], 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**[3] 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**[4], a JavaScript API. It is a parser generator that integrates both lexical and syntactical 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 a 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 is presented in Listing 1. 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.

Listing 1 Grammar.

```
Code = Line* _

Line = (_ Instruction)  ([ \t\r]* Comment)*
     / Comment

Instruction = Label ':'
            / Inst_Atom
            / Inst_Int _ Integer
            / "pushf" _ Float
            / "pushs" _ String
            / "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" / "nop"
          / "atoi" / "atof" / "itof" / "ftoi" / "stri" / "strf"
          / "pushsp" / "pushfp" / "pushgp" / "loadn" / "storen"
          / "swap" / "writei" / "writef" / "writes" / "read" / "call"
          / "return" / "allocn" / "free" / "dupn" / "popn" / "padd"
          / "writeln" / "and" / "or"

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

[2] https://nodejs.org/en/
[3] https://expressjs.com/
[4] https://peggyjs.org/

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 in the form of an array and in each index every instruction code is saved along with its operands. To enable future connection between the code area and the animation, the Machine Code also saves the line number in which the instruction was written on. At the end, the grammar replaces the label addresses in the array with their respective array index.

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

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

In our tool, the Assembler generated by peggy is called each time the user clicks in the `Run` button, receiving as an input the text from the Code Sector and sending its result to the Virtual Machine.

Multiple features were implemented in the various sectors in order to improve this machine as an educational tool. We call *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:**
  - **Clickable Instructions:** by clicking on a code instruction (operator mnemonic), a text box appears with the operation description.
  - **State Choice:** after a complete run of the input program, the user can, by clicking on an instruction line number, select the state corresponding to that instruction; 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:**
  - **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:**
  - **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 interface of the Web application that implements the desired educational version of the virtual machine VM, EWVM, can be observed in Figure 3. As planned, the user can analyse the state of the machine in each step of the code execution, enabling the student to get a clear understanding of the actual effect of each instruction over the components of the machine (this is the so-called *operational semantics* of his input program). For that purpose, the student can observe the values of registers and memory blocks using the information

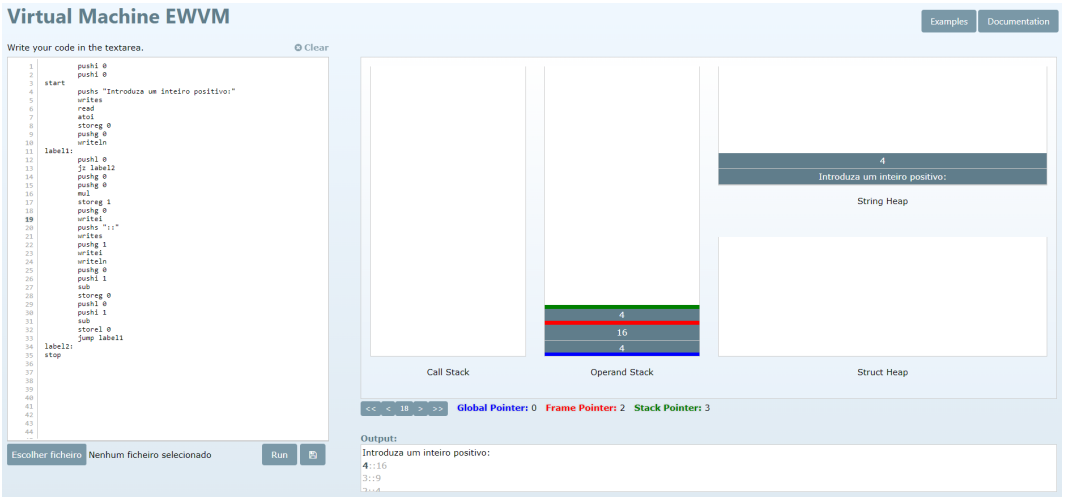

Figure 3 EWVM

displayed in the Animation Sector, while the corresponding instruction is highlighted in the Code Sector. In the Interaction Sector, the text printed is differentiated by colour. In this manner, the user can have a better understanding of the machine's behaviour.

As it is essential to the learning experience, a **Manual** is included containing the code's Documentation, where instructions are listed and explained. In order to give users a little more guidance and help, the website offers **Program Examples** 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.

5 Conclusion

When this work was initiated the aim was to deploy an environment to compile and execute VM programs with all the features described along the paper. We consider that we have attained this goal and that we went a little further. The created environment will be a valuable tool to teach students and lower their learning curve, as it will allow them to learn faster most of the key concepts in code generation. This tool allows teachers to create scenarios and test those scenarios before taking them to the classroom.

EWVM is available in a public URL: `https://ewvm.epl.di.uminho.pt/`.

Although we still do not have sufficient data to sustain these claims, EWVM is now being used in the Language Processing Course with more than 160 students and is receiving very positive feedback. At the end of the next year we will have more data to demonstrate our beliefs.

Concerning future work, one other side goal was to have a tool easy to install. Since EWVM is a web application, once installed all users can access it with a simple browser. However, someone has to install it and to ease that job we will dockerize the application reducing its installation to a command line execution. We also intend to tune the VM language and now we have a platform that will help us doing that. For instance, the VM instruction set has some graphical instructions that, due to many difficulties in the graphical output, have been little explored in the actual tool. As future work we intend to give space to these instructions adding a panel to the interface with an HTML canvas or having the VM compiler produce SVG or other format that browsers can display and animate.

References

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

2 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.

3 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.

4 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.

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

6 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, pages 440–441, New York, NY, USA, 2004. Association for Computing Machinery. doi:10.1145/986537.986647.

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

8 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), January/June 2009.

9 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.

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

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

12 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.

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

14 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. doi:10.1145/1328195.1328197.

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

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