IMPACTO CIENTÍFICO EDITORA

IGUANA MAC INTERPRETER: AN INTERPRETER FOR THE MAC LANGUAGE BY ANDREW S. TANENBAUM

IGUANA MAC INTERPRETER: UM INTERPRETADOR PARA A LINGUAGEM MAC DE ANDREW S. TANENBAUM

IGUANA MAC INTERPRETER: UN INTÉRPRETE PARA EL LENGUAJE MAC DE ANDREW S. TANENBAUM

https://doi.org/10.56238/edimpacto2025.065-004

João Gabriel Freitas Cavalcante[1], Ivan Saraiva Silva[2], Maryane Francisca Araujo de Freitas Cavalcante[3]

ABSTRACT

The article presents the Iguana MAC Interpreter, an educational interpreter for the MAC assembly language proposed by Andrew S. Tanenbaum, developed with the aim of supporting the teaching of Computer Architecture and low-level programming. The interpreter executes MAC code interactively on a 16-bit stack-based architecture, offering support for memory manipulation operations, arithmetic, control flow, bitwise operations, and debugging. Implemented in the Rust programming language, the system adopts a two-pass algorithm for label resolution and incorporates explicit runtime error-handling mechanisms. The tool enables visualization of the machine's internal operation and practical experimentation through classic examples, such as the Hello, World! program. The results indicate that the Iguana MAC Interpreter constitutes a relevant didactic resource for promoting active learning and conceptual understanding of the fundamentals of computer architecture.

Keywords: Assembly Language. Computer Architecture. Educational Interpreter. Active Learning. Computer Science Education.

RESUMO

O artigo apresenta o Iguana MAC Interpreter, um interpretador educacional para a linguagem assembly MAC proposta por Andrew S. Tanenbaum, desenvolvido com o objetivo de apoiar o ensino de Arquitetura de Computadores e programação de baixo nível. O interpretador executa código MAC de forma interativa em uma arquitetura de 16 bits baseada em pilha, oferecendo suporte a operações de manipulação de memória, aritmética, controle de fluxo, operações bitwise e depuração. Implementado na linguagem Rust, o sistema adota um algoritmo de duas passagens para resolução de rótulos e incorpora mecanismos explícitos de tratamento de erros em tempo de execução. A ferramenta possibilita a visualização do funcionamento interno da máquina e a experimentação prática por meio de exemplos clássicos, como o programa Hello, World!. Os resultados indicam que o Iguana MAC Interpreter constitui um recurso didático relevante para promover a aprendizagem ativa e a compreensão conceitual de fundamentos de arquitetura de computadores.

[1] Undergraduated student in Computer Science.
[2] Dr. in Informatics.
[3] Master's student in Intellectual Property.

Palavras-chave: Linguagem Assembly. Arquitetura de Computadores. Interpretador Educacional. Aprendizagem Ativa. Ensino de Computação.

RESUMEN

El artículo presenta el Iguana MAC Interpreter, un intérprete educativo para el lenguaje ensamblador MAC propuesto por Andrew S. Tanenbaum, desarrollado con el objetivo de apoyar la enseñanza de Arquitectura de Computadores y programación de bajo nivel. El intérprete ejecuta código MAC de forma interactiva en una arquitectura de 16 bits basada en pila, ofreciendo soporte para operaciones de manipulación de memoria, aritmética, control de flujo, operaciones bit a bit y depuración. Implementado en el lenguaje de programación Rust, el sistema adopta un algoritmo de dos pasadas para la resolución de etiquetas e incorpora mecanismos explícitos de manejo de errores en tiempo de ejecución. La herramienta permite la visualización del funcionamiento interno de la máquina y la experimentación práctica mediante ejemplos clásicos, como el programa Hello, World!. Los resultados indican que el Iguana MAC Interpreter constituye un recurso didáctico relevante para promover el aprendizaje activo y la comprensión conceptual de los fundamentos de la arquitectura de computadores.

Palabras clave: Lenguaje Ensamblador. Arquitectura de Computadores. Intérprete Educativo. Aprendizaje Activo. Enseñanza de la Computación.

1 INTRODUCTION

Assembly language is a symbolic representation of machine instructions, allowing low-level operations to be expressed in a more intelligible way to humans. The emergence of mnemonic assembly languages in the early 1950s represented a significant advance in the history of programming, by reducing the complexity of direct coding in binary and expanding the accessibility to software development. For this reason, such languages are classified as second-generation languages, falling between machine languages, considered first-generation, and the more abstract levels of programming that would be consolidated later (Patterson; Hennessy, 2005; Aho et al., 2007).

Although the assembly language allows a direct interaction with the hardware, its programming requires the developer to reason in terms close to the operation of the machine, which makes it a laborious, slow and error-prone activity. With the advancement of modern compilers, capable of generating assembly code with performance comparable to that produced by specialists, manual writing in this language has become widely discouraged, mainly due to the longer coding and debugging time, the loss of portability and the difficulties of maintaining the software. Even so, it is noteworthy that the compiled code, resulting from this automated process, tends to present superior performance when compared to the interpreted execution of the same program (Patterson; Hennessy, 2005).

The MAC macro assembly language, also called MAC-1, was conceived by Andrew S. Tanenbaum as a didactic tool for teaching Computer Architecture, characterized by its simplicity and focus on low-level operations. In the context of the Computer Architecture course at the Federal University of Piauí, this language was used as a pedagogical resource for understanding the fundamentals of assembly. However, the lack of a practical tool that would allow the execution and testing of MAC programs evidenced a limitation in the learning process, by restricting the experimentation of the concepts addressed.

In view of this gap, the motivation arose for the development of an interpreter capable of facilitating interaction with the MAC language and expanding its educational applicability. The Iguana MAC Interpreter was designed precisely to meet this need, enabling the direct execution of MAC programs and promoting a more practical and interactive approach to teaching low-level programming.

Implemented in the Rust language, the Iguana MAC Interpreter executes the MAC source code incrementally, without the need for prior compilation. The system operates on a stack-based 16-bit architecture, with a fixed capacity of 32,768 items, and supports an

extended set of instructions, covering constant load operations, memory manipulation, arithmetic, flow control, bitwise operations, debugging, and custom functionality. In addition, the interpreter extends the original format of the MAC instructions from 16 to 24 bits, assigning 8 bits to the operation code and 16 bits to the argument, which gives greater flexibility and expressiveness to the language.

Thus, this work aims to present the Iguana MAC Interpreter as an educational tool, describing in a synthetic way its conception, implementation and practical use in the teaching of assembly programming. Through classic examples, such as the printing of the message *"Hello, World!"*, seeks to highlight its pedagogical applicability. In addition, the introduction delimits the scope of the subsequent sections, in which the interpreter architecture, its set of instructions, examples of use and educational benefits are addressed, emphasizing its relevance as a resource to support learning in Computer Architecture.

2 THEORETICAL FRAMEWORK

2.1 ASSEMBLY LANGUAGES, COMPUTER ARCHITECTURE, AND INTERPRETERS

Assembly languages play a central role in understanding the hardware-software interface, since they symbolically represent the instructions executed directly by the processor. As discussed by Patterson and Hennessy, the study of this level of abstraction makes it possible to understand the inner workings of computer architecture, including the execution model, memory organization, and instruction flow control.

In this context, didactic languages such as MAC play a relevant role in enabling the conceptual exploration of simplified architectures, making the internal mechanisms of operation of computer systems more accessible. By reducing the complexity inherent to real architectures, these languages favor the understanding of the fundamental principles that govern the interaction between software and hardware, as well as the processes of translation and execution of instructions, contributing to the conceptual formation in computer architecture and low-level programming.

The hardware-software interface is a fundamental axis in studies of computer organization and design, by examining the interaction between physical components, such as CPU and memory, and the software executed in the system. As presented by Patterson and Hennessy in *Computer Organization and Design: The Hardware/Software Interface*, understanding this interaction is essential to analyze the performance, efficiency, and

behavior of programs, since the way software exploits hardware resources directly influences the execution and optimization of computer systems (Patterson; Hennessy, 2005).

Assembly language is a symbolic representation of machine instructions, classified as second-generation, succeeding machine language. The development of assembly mnemonics has made programming easier, making it more accessible, but it still requires the programmer to "think like the machine". Although it is essential for computer architecture, the sources consulted do not mention the didactic language "MAC" in the context of language processing (Aho et al., 2007).

From the point of view of language processing, interpreters stand out as pedagogical tools for enabling the incremental execution of the code, allowing the explicit observation of the internal state of the machine during execution. Unlike compilers, which perform the complete translation of the program before its execution, interpreters make the computational process more transparent, by highlighting steps such as the evaluation of instructions, stack manipulation, and label resolution, fundamental aspects for understanding low-level programming and computer architecture.

Compilers and interpreters therefore adopt different execution strategies. While compilers translate the source program into a target language, usually producing machine code with higher execution efficiency, interpreters appear to directly execute the operations specified in the source code, processing them instruction by instruction. That difference entails specific advantages and limitations, in particular as regards the performance and clarity of the enforcement process.

From this perspective, the pedagogical relevance of interpreters lies mainly in their diagnostic capacity, since the step-by-step execution of the program allows for a more accurate identification of errors and an understanding of their impact on the state of the machine. In addition, contemporary compilation approaches often adopt hybrid models, such as the generation of intermediate code (e.g., *bytecodes*), which is then interpreted by a virtual machine, reconciling the benefits of translation with portability and reinforcing the conceptual importance of interpretation in the teaching of programming languages (Aho et al., 2007).

2.2 EDUCATIONAL TOOLS, SYSTEMS LANGUAGES AND ACTIVE LEARNING

The development of educational tools aimed at teaching computing requires the adoption of programming languages that reconcile low-level control, performance and reliability. In this context, the Rust language has stood out for offering guarantees of memory

security and concurrency without compromising efficiency, characteristics especially relevant for the implementation of interpreters and robust educational systems. In addition, its use contributes to bringing students closer to contemporary practices of systems programming, aligning teaching with technologies widely used in the area.

At the same time, the design of these tools is associated with the adoption of pedagogical approaches that favor active learning, in which students participate directly in the process of knowledge construction. The combination of languages suitable for the development of low-level systems and active educational methodologies enhances the understanding of complex concepts, by transforming abstract content into practical and interactive experiences in the teaching of computing (Freeman et al., 2014).

The use of interactive educational tools is aligned with the principles of active learning, a pedagogical approach that is opposed to the traditional model centered on the passive transmission of content. From this perspective, the student assumes a more participatory role in the learning process, through experimentation, problem solving and direct interaction with the objects of study, which favors a more meaningful understanding of the concepts.

Empirical evidence from meta-analysis studies indicates that active learning produces consistent positive impacts on academic performance in science, technology, engineering, and mathematics (STEM) courses. Among the main results observed are the average increase in student performance in assessments, as well as the reduction in failure rates when compared to classes submitted exclusively to traditional lectures (Freeman et al., 2014).

Thus, it is highlighted that the benefits of active learning are manifested in a broad way, reaching different disciplines and educational contexts, with particularly expressive effects in smaller classes. In this sense, the adoption of interactive tools in the teaching of computing reinforces pedagogical practices that promote greater student engagement and better academic performance, contributing to the consolidation of a more effective and participatory education.

From this perspective, the Rust language has consolidated itself as a relevant option in the context of education and systems development, by meeting the demand for programming languages that combine performance, reliability, and low-level control. As an open-source language for systems programming, Rust enables the construction of efficient and robust software, fundamental characteristics for both real applications and educational tools (Patterson; Hennessy, 2005; Klabnik; Nichols, 2018).

Klabnik and Nichols (2018) cite that Rust's suitability for low-level educational and programming systems stems, to a large extent, from its balance between control and ergonomics. The language provides direct access to critical aspects such as memory management, while incorporating high-level abstractions that reduce the complexity traditionally associated with systems languages. This balance makes the development process safer and more understandable, especially in teaching contexts.

Another central aspect of Rust refers to the memory and concurrency security guarantees provided by the compiler, through the *ownership* and *borrowing mechanisms*. These features make it possible to prevent common errors, such as invalid memory accesses and race conditions, without resorting to a garbage collector. In addition, the use of zero-cost abstractions ensures that the generated code maintains high performance, reinforcing the idea that security and efficiency can coexist in modern systems languages.

In addition, the adoption of Rust in educational environments contributes to bringing students closer to contemporary system programming practices, while favoring the understanding of fundamental concepts, such as memory management, concurrency, and computational efficiency. In this way, the use of Rust not only supports the development of robust educational tools but also prepares students for real challenges in the field of systems computing (Freeman et al., 2014).

Therefore, in core disciplines of Computer Science and Engineering, such as Computer Architecture and Computer Organization and Design, the use of interactive tools, such as educational interpreters, is particularly relevant because it enables the direct exploration of the interaction between hardware and software. By making visible the mechanisms that sustain the efficient execution of programs, these tools contribute to the conceptual consolidation of the contents and to a more integrated understanding of the computational foundations (Patterson; Hennessy, 2005).

3 METHODOLOGY

The study is characterized as a research of applied nature, with a qualitative and descriptive approach, aimed at the development and analysis of an educational computational tool. The work focused on the design, implementation and functional evaluation of the Iguana MAC Interpreter, an interpreter for the MAC assembly language, with emphasis on its applicability in the teaching of computer architecture and low-level programming.

The methodology adopted initially comprised a conceptual and technical review of assembly languages, interpreters and computer architecture, based on classical and contemporary literature, especially the work of Andrew S. Tanenbaum, which underlies the MAC language used in this study. This stage subsidized the definition of the functional requirements of the interpreter and the preservation of the didactic principles of the original language.

Then, the Interpreter was developed, implemented in the Rust language, following an incremental approach. The system architecture was defined based on a 16-bit stack-oriented machine, including data memory, instruction memory, accumulator, stack pointer, and program counter. The interpretation process adopted a two-pass algorithm, the first being intended for the construction of the table of symbols and the second for the execution of the instructions, with label resolution and flow control.

The validation of the tool occurred through functional tests, using programs written in the MAC language, with emphasis on the classic example of printing the string *"Hello, World!".* These tests made it possible to verify the correct functioning of the instructions, the management of the stack, the memory manipulation and the handling of errors at runtime, ensuring the conformity of the interpreter with the proposed specification.

Finally, the analysis of the interpreter was carried out from an educational perspective, considering its operational clarity, ability to visualize low-level concepts and potential to support the teaching-learning process in Computer Architecture disciplines. No quantitative experiments were carried out with users, since the objective of the study is the presentation and technical discussion of the tool, and not the statistical measurement of pedagogical performance.

4 RESULTS

4.1 INTERPRETER ARCHITECTURE

Iguana MAC Interpreter is designed with an architecture that reflects the principles of simplicity and efficiency, adapting Andrew S. Tanenbaum's MAC language for interactive execution in a 16-bit environment. One of the central aspects of this architecture is its modular approach, divided into distinct components that manage data memory, instruction memory, and runtime processing, all implemented in the Rust language to ensure robustness and security.

The instruction memory is organized as a dynamic vector, where each position represents an instruction composed of an 8-bit operation code (CODOP) and a 16-bit argument, totaling a 24-bit format. This choice differs from the original 16-bit format of MAC and allows for the inclusion of additional instructions and a wider range of arguments, such as memory addresses or constant values. In each position, there are also two auxiliary fields, both 32-bit, called row and column, which store the position of each token of the source code and can be later used for debugging errors at runtime. During execution, the interpreter traverses this dynamic vector, using a Program Counter (PC) to track the current instruction, which provides flexibility in manipulating control flows, such as jumps and subroutine calls.

Another key element is the use of a two-pass algorithm to resolve references to *labels*. In the first pass, the source code is parsed to construct a table of symbols, implemented as a *HashMap* in Rust, which associates each *label* with a memory address (in the .data segment) or an instruction line (in the .text segment). On the second pass, the statements are processed, replacing the references to *labels* with the corresponding addresses. This approach allows the interpreter to efficiently resolve references, even in cases where statements can be reordered or where *labels* appear before they are defined.

The interaction between the accumulator (ac) and the stack is managed by a *Stack Pointer* (sp) that operates in a top-down manner, decrementing when stacking values and incrementing when unstacking them. The stack, with a fixed capacity of 32,768 16-bit items, is allocated in a contiguous region of memory, and its downward growth direction is complemented by specific instructions, such as INSP (increments sp by subtracting) and DESP (decrements sp by adding), which adjust the pointer with respect to its mathematical behavior. This design reflects a didactic adaptation that makes it easier to visualize the behavior of the stack in an educational context.

Finally, the architecture incorporates a runtime error handling system, such as the detection of values outside the allowed range (e.g. [ERROR] Value range exceeded (-32768...32767) [LINE: 6, COL: 5]), which is triggered when the accumulator tries to store a value outside the 16-bit range. This approach not only improves the robustness of the interpreter but also serves as a valuable educational tool, allowing students to better understand the limits and characteristics of the 16-bit architecture.

4.2 SET OF INSTRUCTIONS

The Iguana MAC Interpreter instruction set is an extension of the original MAC language proposed by Andrew S. Tanenbaum, being developed to meet the demands of an interactive educational environment. While the traditional MAC language focuses on the basic operations of a 16-bit architecture, Iguana incorporates additional instructions that extend its functionality, especially in terms of debugging and usability, without compromising the conceptual simplicity of the original language.

This expansion is designed to balance didactic rigor and functional features, allowing users to explore the fundamental concepts of assembly programming while more transparently tracking the program's behavior during execution. Direct interaction with the interpreter favors the understanding of the machine's internal mechanisms, contributing to a more active and meaningful learning.

The format adopted by Iguana allows the inclusion of operations that do not exist in the base language, such as PRINTLNAC and SLEEPI, which support viewing the state of the program and controlling the temporal flow of execution. These instructions are designed to operate in a manner consistent with the stack-based architecture of the interpreter, ensuring consistency between the execution model and the set of operations available.

The Iguana MAC Interpreter instruction set covers the following main categories: loading constants (e.g. LOCO, to initialize values in the accumulator), memory operations (e.g. LODD, to load values from memory), arithmetic operations (e.g. ADDD, to sum values), flow control (e.g. JUMP, to divert execution), bitwise operations (e.g. ANDI, to perform logical AND), debugging (e.g. PRINTLNAC, to display the accumulator value) and miscellaneous operations (e.g. HALT, to stop execution). These instructions expand on the functionalities of the original MAC language, making it easier to manipulate data, control execution, and debug.

Iguana's instruction set not only preserves the essence of the original MAC language, but also enriches it with features that make learning more dynamic and accessible. Debugging operations, for example, allow users to inspect the internal state of the interpreter in real time, while custom instructions, such as SLEEPD and SLEEPI, introduce temporal control, useful in simulations or interactive demonstrations.

4.3 USAGE EXAMPLE: HELLO, WORLD!

To illustrate how the Iguana MAC Interpreter works, a classic example of programming is presented: the printing of the string *"Hello, World!"* on the screen. This program

demonstrates the use of memory manipulation, flow control, and debugging operations in a practical context, making it an ideal use case for beginners in assembly study. The code, written in the MAC language adapted by Iguana, is divided into two main sections: the data declaration (.data) and the executable code (.text). Below, the program is presented and analyzed in detail.

Figure 1

```
.data
        STRING: .asciiz "Hello, World!"  # initializes the string
.text
                LOCO STRING              # ac = STRING as a pointer
                SWAP                     # ac <-> sp
        LOOP:
                LODL 0                   # ac = *sp
                JZER END                 # if ac == 0 goto END
                PRINTACCHAR              # print ac as a char
                INSP 1                   # sp = sp - 1
                JUMP LOOP                # goto LOOP
        END:
                HALT                     # finishes the program
```

4.4 CODE ANALYSIS

In the .data section, a *label* is defined: STRING, which uses the .asciiz directive to allocate the string *"Hello, World!"* in memory with a null terminator (value 0). This declaration is processed by the interpreter, which associates the *label* with a memory address in the symbol table.

The program starts with the LOCO STRING instruction, which loads the base address of the STRING string into the accumulator (ac). SWAP then swaps the accumulator value with the top of the stack, stored on the *Stack Pointer* (sp), initializing sp as a pointer to the string. This step is crucial because it allows the stack to point to the beginning of the string to be printed.

The main block of the program is a loop identified by the *LOOP label*. The LODL 0 instruction loads into the accumulator the value pointed to by sp (the current character of the string). The JZER END statement checks to see if this value is zero, i.e., the null terminator of the string, and if so, diverts execution to the *END label*, terminating the loop. Otherwise, PRINTACCHAR prints the accumulator value as an ASCII character, displaying it on the screen. The INSP 1 instruction decrements the *Stack Pointer* by 1 (subtracting

mathematically, due to the downward growth of the stack), advancing to the next character. Finally, JUMP LOOP returns to the beginning of the loop, repeating the process until the null terminator is found. Upon reaching END, the HALT statement terminates the execution of the program.

4.5 EXECUTION AND RESULT

When run with the runiguana program.asm command, the interpreter processes the code and prints *"Hello, World!"* in the standard output, character by character, with no additional line breaks, as specified by PRINTACCHAR. This result demonstrates the Iguana's ability to manipulate strings in memory and use flow control and debugging instructions efficiently. The use of the null terminator as a stop condition reflects a common practice in low-level languages, while the stack-based operation highlights the interpreter architecture.

This example is particularly valuable in an educational context, as it introduces fundamental assembly concepts such as the use of pointers (sp), conditional loops (JZER, JUMP), and memory manipulation (LODL).

4.6 EDUCATIONAL BENEFITS

The Iguana MAC Interpreter was conceived as an educational tool to make up for the absence of practical resources in Andrew S. Tanenbaum's MAC language teaching, and its pedagogical benefits are evident both for beginner students and for those who seek to deepen their knowledge at a low level. The interpreter converts abstract theoretical concepts into practical experiences by allowing you to write, test, and debug programs in assembly, promoting a more solid understanding of low-level fundamentals. The expanded set of instructions also plays a crucial role in enriching the educational process.

The inclusion of custom operations broadens the scope of hands-on exercises students can undertake. In addition, Iguana fosters active learning by encouraging students to develop their own programs and test hypotheses. The stack-based architecture, with its simplicity and well-defined limitations, serves as a teaching model that reflects actual hardware constraints, helping students internalize memory management concepts. The run-time error handling system, which flags conditions as out-of-range values, reinforces this learning by providing immediate, contextualized feedback, turning errors into teaching opportunities.

In short, the Iguana MAC Interpreter stands out as a valuable pedagogical resource for its ability to connect theory and practice, promote experimentation, and adapt to the needs of a modern educational environment. Whether you're introducing the fundamentals of assembly programming or exploring more advanced topics, the interpreter provides a solid foundation that fosters students' interest and proficiency in computer architecture, aligning with Tanenbaum's original goal of creating an accessible language for teaching.

5 CONCLUSION

This work presented the Iguana MAC Interpreter, an educational interpreter for the MAC assembly language proposed by Andrew S. Tanenbaum, designed with the objective of making up for the absence of practical tools for the execution and experimentation of this language in the teaching of Computer Architecture. Throughout the article, the theoretical context that underlies the use of low-level languages in the training process was detailed, the methodology adopted for the development of the interpreter and the description of its architecture, set of instructions and operation, evidencing the adherence of the tool to the didactic principles of the original language.

The results demonstrate that the Iguana MAC Interpreter enables the interactive execution of MAC programs in a stack-based 16-bit architecture, offering additional features that extend its educational utility, such as debugging instructions, temporal control, and runtime error messages. The use of a two-pass algorithm for label resolution, combined with an explicit error handling system, contributes to making visible central aspects of the machine's internal workings, favoring the understanding of concepts such as memory manipulation, flow control and architectural limits.

From a pedagogical point of view, the interpreter stands out for promoting active learning, by allowing students to write, execute and debug programs in assembly, transforming abstract theoretical concepts into practical experiences. The simplicity of the adopted architecture, combined with the operational clarity of the interpreter, enables the gradual exploration of fundamental low-level programming content, aligning with the educational purpose that motivated the creation of the MAC language.

As limitations, it is noteworthy that this study did not carry out quantitative empirical evaluations with users, focusing on the presentation and technical analysis of the tool. Future works may contemplate the application of the Iguana MAC Interpreter in classroom contexts, the expansion of its set of instructions and the integration with graphic resources or interactive

educational environments, deepening its contribution to the teaching of Computer Architecture and low-level languages.

REFERENCES

Aho, A. V., Lam, M. S., Sethi, R., & Ullman, J. D. (2007). Compilers: Principles, techniques, and tools (2nd ed.). Pearson Addison-Wesley.

Ball, T. (2018). Writing an interpreter in Go. Leanpub.

Freeman, S., Eddy, S. L., McDonough, M., Smith, M. K., Okoroafor, N., Jordt, H., & Wenderoth, M. P. (2014). Active learning increases student performance in science, engineering, and mathematics. Proceedings of the National Academy of Sciences of the United States of America, 111(23), 8410–8415. https://doi.org/10.1073/pnas.1319030111

Klabnik, S., & Nichols, C. (2018). The Rust programming language. No Starch Press.

Nystrom, R. (2021). Crafting interpreters. https://craftinginterpreters.com/

Patterson, D. A., & Hennessy, J. L. (2005). Computer organization and design: The hardware/software interface (3rd ed.). Morgan Kaufmann.

Tanenbaum, A. S. (1992). Organização estruturada de computadores (3ª ed.). LTC.