Bytecode Virtual Machine
Technical Report

January 7, 2026

1 Introduction

Virtual machines form the backbone of many modern execution environments by providing an abstraction layer between software and hardware. This project focuses on the design and implementation of a custom bytecode-based virtual machine (VM) along with a corresponding assembler.

The VM follows a stack-based execution model and executes programs written in a custom assembly language. These assembly programs are translated into compact binary bytecode by the assembler and subsequently executed by the virtual machine. The project demonstrates core concepts of low-level system design, including instruction encoding, program counter management, memory modeling, stack-based computation, control flow handling, and function call mechanisms.

1.1 Objective

The primary objective of this project is to build a complete bytecode execution pipeline starting from assembly source code and ending with deterministic runtime execution. The system is intended to be minimal yet expressive enough to support arithmetic computation, branching, looping constructs, memory access, and function calls.

1.2 Design Methodology

A modular and layered design methodology was adopted. The assembler and the virtual machine are implemented as separate components with clearly defined responsibilities. The assembler handles all source-level concerns such as parsing, syntax validation, and label resolution, whereas the VM focuses purely on bytecode execution and runtime correctness.

This separation of concerns improves maintainability, simplifies debugging, and allows each component to be tested independently.

2 Bytecode Generator

The assembler serves as the bytecode generator and converts human-readable assembly programs into binary bytecode files. These bytecode files contain no textual markers and are interpreted solely based on byte order and instruction format by the virtual machine.

2.1 Two-Pass Assembly Process

The assembler employs a two-pass strategy to handle forward-referenced labels:

- **Pass 1 (Symbol Resolution)**: During the first pass, the assembler scans the source file to identify all labels and computes their corresponding bytecode addresses. This pass builds a symbol table but does not generate any bytecode.
- **Pass 2 (Code Generation)**: In the second pass, the assembler translates each instruction into its binary representation. When a label reference is encountered, the previously computed address is retrieved from the symbol table and embedded into the bytecode.

This approach ensures correct resolution of both forward and backward jumps and is commonly used in real-world assemblers.

2.2 Binary Encoding and Endianness

The generated bytecode is designed to be compact and efficient. No whitespace or separators are used. Instead, the VM relies entirely on predefined instruction widths to decode the byte stream.

2.2.1 Instruction Formats

The instruction set consists of two distinct formats:

- **Type A Instructions (1 Byte)**: These instructions operate entirely on values already present on the operand stack and require no additional data. Examples include ADD, SUB, POP, DUP, and HALT.
- **Type B Instructions (5 Bytes)**: These instructions require an immediate operand. The first byte stores the opcode, followed by a 4-byte signed integer operand. Examples include PUSH, LOAD, STORE, JMP, JZ, JNZ, and CALL.

2.2.2 Endianness

All operands are encoded using little-endian format. This choice aligns with the native byte order of modern x86 processors and ensures consistent decoding across platforms.

2.2.3 Operand Reconstruction

Rather than casting raw memory into integers, the VM reconstructs operands manually using bitwise shifts and OR operations. This ensures portability and avoids undefined behavior related to memory alignment.

3 Architecture of the Virtual Machine

The virtual machine simulates a simplified CPU using a stack-based execution model. Unlike register-based architectures, all computation in this VM occurs via the operand stack.

3.1 Memory Model

The VM memory is divided into three clearly separated regions:

- **Code Memory**: Stores the bytecode program. This memory is read-only during execution.
- **Data Memory**: A fixed-size array of 256 integer locations used for persistent storage through LOAD and STORE instructions.

- **Operand Stack**: A fixed-size stack capable of holding up to 1024 integer values. This stack is used for all arithmetic operations and intermediate results.

This explicit separation simplifies execution semantics and prevents unintended interference between program code and runtime data.

3.2 Registers

The VM uses a minimal set of control registers:

- **Program Counter (PC)**: Tracks the index of the next instruction to be executed within the bytecode array.
- **Stack Pointer (SP)**: Tracks the current top of the operand stack.

4 Instruction Dispatch Strategy

The VM executes instructions using a classical fetch–decode–execute cycle.

4.1 Fetch and Decode

In each iteration of the execution loop, the VM fetches the opcode pointed to by the program counter. If the opcode corresponds to a Type B instruction, the VM reads the following four bytes and reconstructs the operand.

The program counter is advanced deterministically based on instruction width, ensuring correct sequential execution.

4.2 Dispatch Mechanism

Decoded instructions are executed using a switch-case dispatch mechanism. Each case implements the semantics of a single instruction and includes runtime safety checks such as stack overflow, stack underflow, invalid memory access, and division by zero.

This approach provides predictable control flow and simplifies debugging and extension.

5 Design of Call Frames

To support subroutines and recursive execution (such as those required for computing factorials), the virtual machine implements a dedicated *return stack*. This separate stack is used exclusively to store return addresses during function calls, ensuring reliable control flow management.

5.1 Function Calls (CALL `0x40`)

When a CALL instruction is executed, the virtual machine performs the following steps:

1. The address of the instruction immediately following the CALL instruction is calculated and treated as the return address.
2. This return address is pushed onto the return stack.
3. The program counter (PC) is updated to the target address of the called function.

5.2 Function Returns (RET `0x41`)

When a RET instruction is executed, control returns to the calling context through the following steps:

1. The address at the top of the return stack is popped.
2. The program counter is set to this popped address, allowing execution to resume exactly from where it was paused.

The use of a dedicated return stack simplifies the overall architecture by preventing operations on the data (operand) stack from corrupting return addresses. This design ensures correct execution of nested and recursive function calls.

6 Benchmark Programs

A comprehensive suite of benchmark programs was developed to validate the correctness and robustness of the VM. These programs cover arithmetic computation, simple and nested loops, memory access patterns, function calls, and iterative algorithms such as factorial and exponentiation.

7 Limitations and Enhancements

7.1 Current Limitations

The current implementation intentionally adopts several constraints to keep the VM simple and deterministic:

- Fixed data memory size of 256 integers.
- Fixed operand stack size of 1024 integers.
- No dynamic memory allocation or heap management.
- Direct memory addressing only.
- Support limited to 32-bit signed integers.
- No built-in input or output instructions.
- Single-threaded execution model.

7.2 Future Enhancements

Several enhancements can be considered to improve expressiveness:

- Indirect addressing modes for array access.
- Dynamic memory allocation mechanisms.
- Support for additional data types.
- Input and output instructions.
- Stack frame pointer for local variables and arguments.

8 Conclusion

This project demonstrates the complete design and implementation of a bytecode virtual machine and assembler. The system successfully executes a wide range of programs using a clean stack-based architecture and deterministic execution model.

The modular design, clear instruction semantics, and explicit handling of control flow and memory make the VM suitable for educational exploration of virtual machine and runtime system design.