Technical Report: Lab 4 - Bytecode Virtual Machine Implementation

Author: Jaskaran Singh

Date: 8 January 2026

1. Introduction

This report documents the design, implementation, and performance characteristics of a custom stack-based **Bytecode Virtual Machine (VM)**. The project consists of two core components: a Virtual Machine written in C for high-performance execution, and an Assembler written in Python for converting human-readable assembly into optimization-ready bytecode.

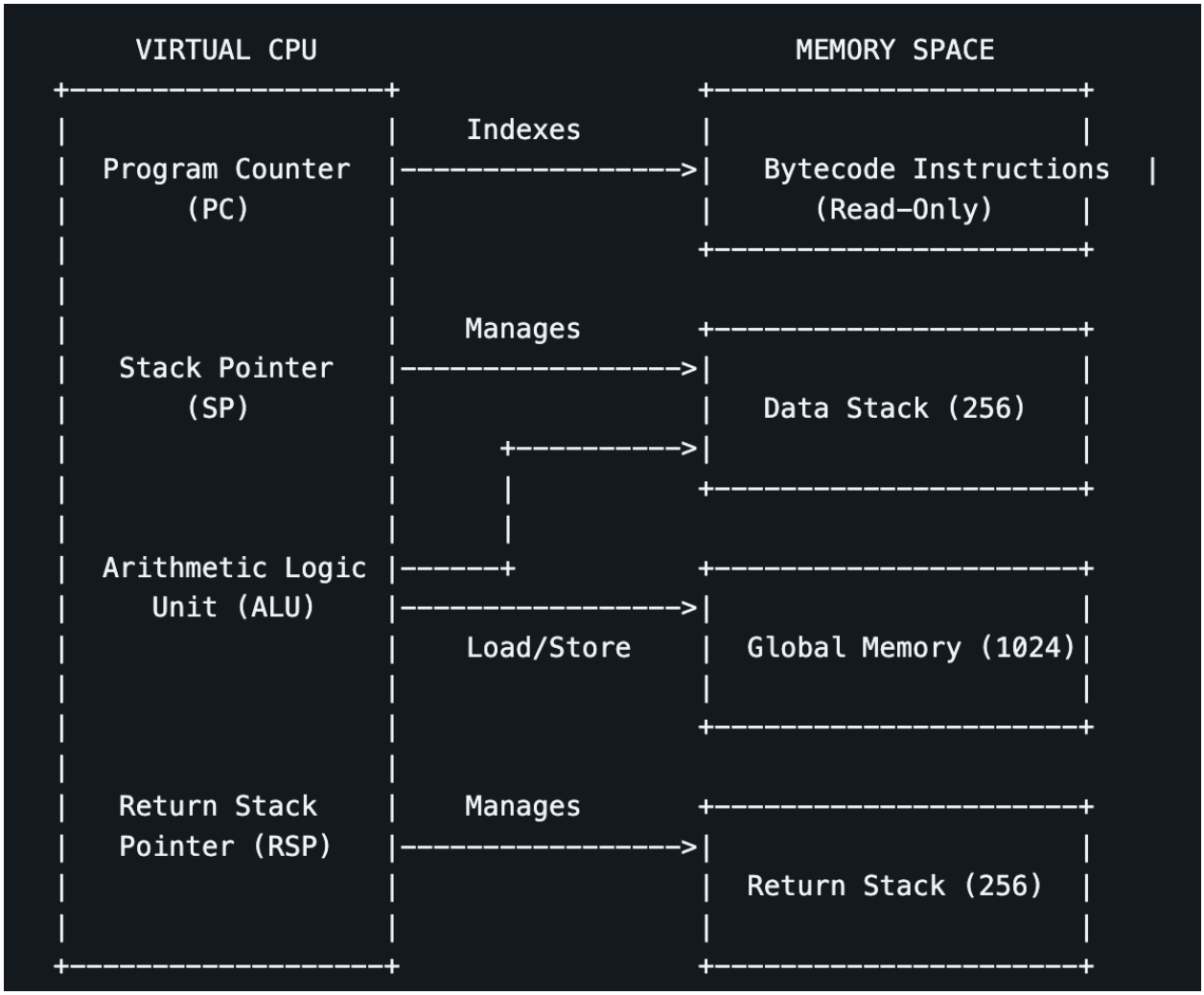
The system is designed to meet requirements for deterministic execution, robust error handling, and a Turing-complete instruction set supporting arithmetic, branching, and function calls.

2. System Architecture

The VM architecture follows a **Harvard-like** model where code (bytecode) and data (stack/memory) are logically separated, although they share the host process's address space.

2.1 Memory Model

The following diagram illustrates the VM's memory organization, highlighting the separation between executable code, working data, and control flow storage.



The VM utilizes four distinct memory regions:

- **Data Stack (stack[256]):**
  - **Type:** int32_t array.
  - **Purpose:** The primary workspace for all arithmetic and logic operations. Operands are popped from here, and results are pushed back.
  - **Design Choice:** A fixed size of 256 words was chosen to fit within CPU cache lines for speed, with explicit bounds checking to prevent overflows.
- **Return Stack (return_stack[256]):**
  - **Type:** uint32_t array.
  - **Purpose:** Stores return addresses (PC + 1) during function calls.
  - **Design Choice:** Separating logic data from control flow data is a critical architectural decision. It prevents buffer overflow attacks where data operations might accidentally overwrite return addresses (a common vulnerability in x86/Von Neumann architectures).

- **Global Memory (memory[1024]):**
    - **Type:** int32_t array.
    - **Purpose:** Simulates Random Access Memory (RAM) for storing variables across function scopes. Accessed via LOAD and STORE instructions.
- **Bytecode Storage (code):**
    - **Type:** uint8_t dynamic array.
    - **Purpose:** Read-only storage for the executable program. The Program Counter (PC) indexes directly into this array.

2.2 CPU Registers & State

The VM maintains minimal state to reduce context-switching overhead:

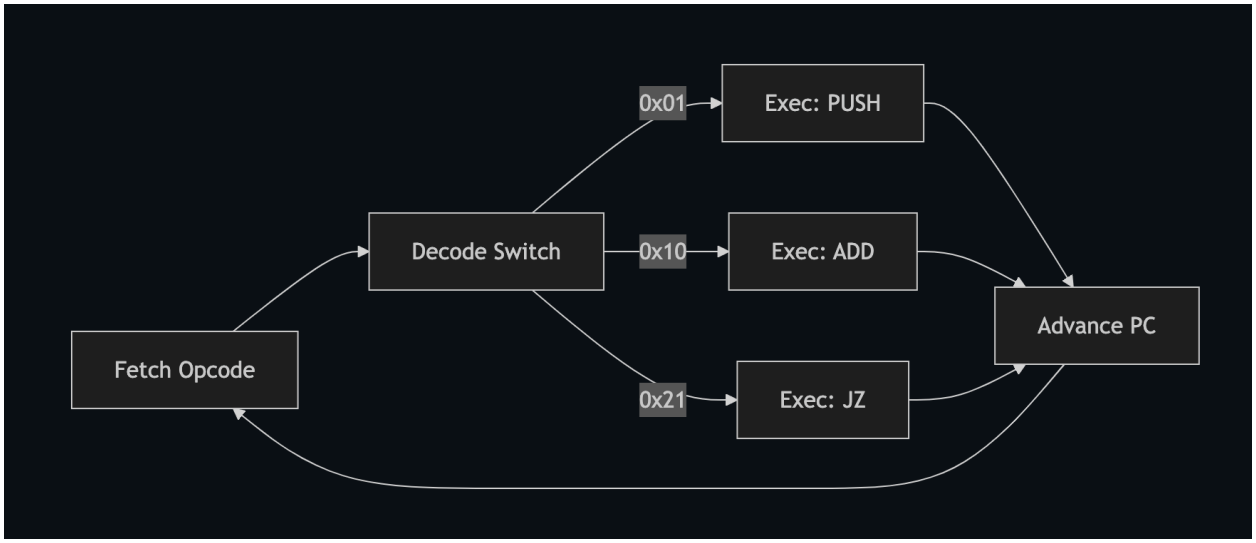
- **PC (Program Counter):** Index of the next instruction to execute.
- **SP (Stack Pointer):** Points to the next free slot on the Data Stack (grows upward).
- **RSP (Return Stack Pointer):** Points to the next free slot on the Return Stack.
- **Running Flag:** Controls the main execution loop.
- **Error Flag:** Indicates if a runtime exception (e.g., Division by Zero) has occurred.

3. Core Mechanisms

3.1 Instruction Dispatch Strategy

The VM employs a **Direct Dispatch** strategy using a central **switch-case** loop inside the **run_vm** function.

The Fetch-Decode-Execute Cycle:

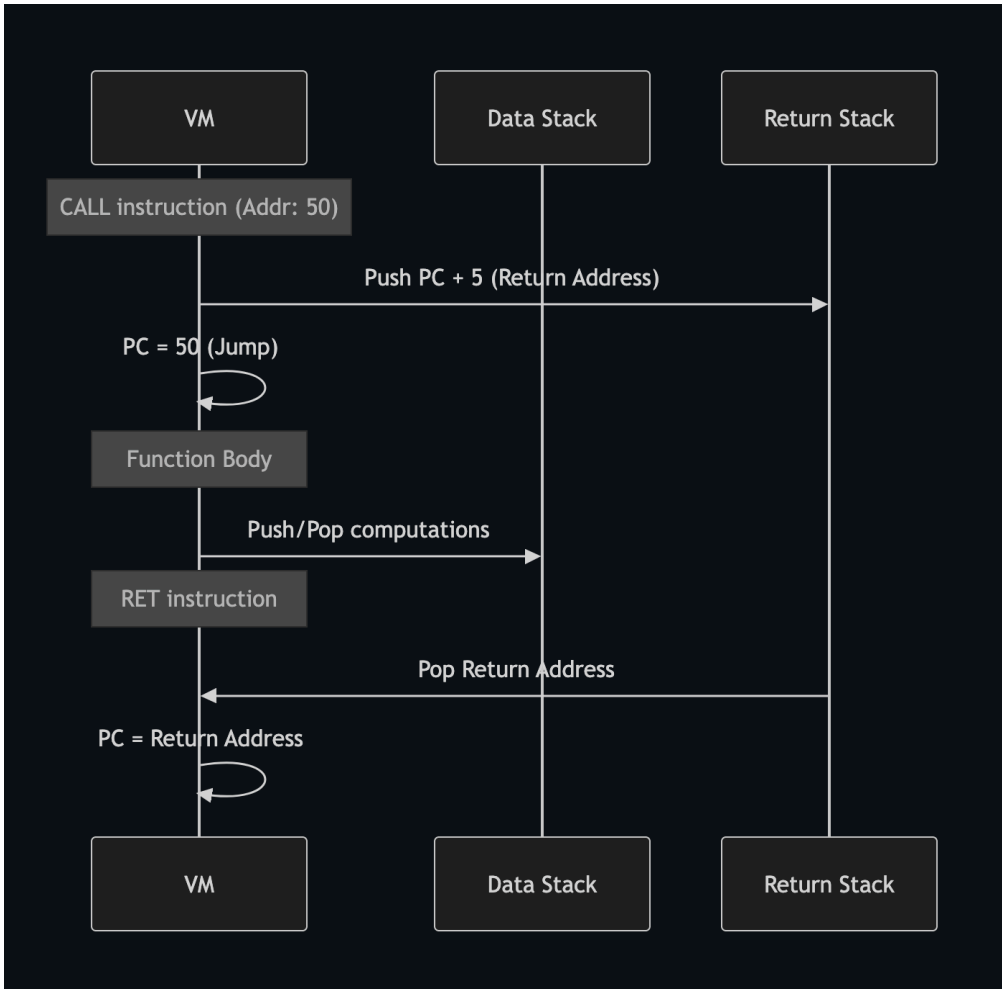


1. **Fetch:** The byte at **code[pc]** is read.
2. **Decode:** The byte serves as the **case** label in the switch statement.
3. **Execute:** The corresponding C code block executes (e.g., performing a **PUSH**).
4. **Advance:** The **PC** is incremented (either by 1 for simple opcodes, or by 5 for opcodes with 32-bit arguments).

Rationale: While threaded code or JIT (Just-In-Time) compilation offers higher performance, a giant switch statement provides the best balance of code simplicity, portability, and "good enough" performance for an educational VM.

3.2 Function Call & Return Mechanism

Functional calls are implemented using a dedicated **Return Stack Frame** mechanism, distinct from the data stack.



The CALL <addr> Instruction:

1. The VM calculates the return address: **current_pc + 5** (1 byte opcode + 4 bytes argument).
2. This address is pushed onto the **Return Stack**.
3. The **PC** is set to **<addr>**, causing an immediate jump.

The RET Instruction:

1. The return address is popped from the **Return Stack**.
2. The **PC** is set to this address.

Advantages:

- Simplifies stack management (no need to calculate offsets to skip over return addresses).
- Enables deep recursion (up to 256 levels) without complex frame pointer logic.

4. Assembler Design

The assembler (**assembler.py**) implements a **Two-Pass** compilation process to handle forward references (calling a function or jumping to a label defined later in the file).

- **Pass 1 (Symbol Resolution):**
    - Scans the source file line-by-line.
    - ignores instructions but calculates their byte size (1 byte for opcodes, 5 bytes for **PUSH**/**JMP** variants).
    - Records declarations of **Labels** (e.g., **LOOP:**) into a Symbol Table **{ "LOOP": 15 }**.
- **Pass 2 (Code Generation):**
    - Re-scans the source file.
    - Translates mnemonics to hex opcodes.
    - Resolves label arguments using the Symbol Table.
    - Writes binary data to the output file.

5. Performance Analysis

Benchmarks performed using a 10-million iteration loop (**benchmark/loop.asm**) on a standard environment yielded the following results:

- **Throughput:** ~133 Million Instructions Per Second (MIPS).
- **Time:** 0.30 seconds for 40,000,000 instructions.

This performance confirms that the C implementation is highly efficient, with minimal overhead per instruction.

6. Limitations and Future Enhancements

6.1 Limitations

1. **Fixed Stack/Memory Size:** Recursion depth is hard-limited to 256 calls. This works for embedded-like tasks but fails for massive algorithms.
2. **Integer-Only Arithmetic:** The VM lacks Floating Point Unit (FPU) support.
3. **Primitive I/O:** The only output mechanism is printing the stack at termination. No **print** or **input** instructions exist.

6.2 Enhancement Proposals

1. **Dynamic Memory (Heap):** Implement **ALLOC** and **FREE** instructions to manage a dynamic heap in the **memory[]** array for string/array processing.
2. **Just-In-Time (JIT) Compilation:** Replace the switch-dispatch with a JIT that maps VM bytecodes directly to x86/ARM machine code for 10x-50x speedups.
3. **Foreign Function Interface (FFI):** Allow **CALL** instructions to invoke native C functions (e.g., **sin()**, **fopen()**) to extend capabilities.