Lab 4: Design and Implementation of a Stack-Based Bytecode Virtual Machine

Jithendra Gannavarapu (2025MCS2743)
Manmadha Rao Kuttum (2025MCS2101)

January 7, 2026

Abstract

This report documents the design and implementation of a stack-based Virtual Machine (VM) and a corresponding two-pass Assembler. The system creates a complete execution toolchain: the Assembler translates human-readable assembly mnemonics into a compact 32-bit bytecode, and the VM executes this bytecode using a secure Fetch-Decode-Execute cycle. Key architectural decisions include a shared instruction definition file for consistency, a two-pass resolution strategy for forward jumps, and a robust runtime environment that safeguards against stack underflows and invalid memory access.

1 Overview

The objective of this lab was to understand the low-level operations of computer systems by building a Virtual Machine (VM) from scratch. Unlike high-level interpreters, this VM operates on a strictly defined bytecode format, requiring manual management of the Program Counter (PC), Operand Stack, and Call Stack.

The project implements a complete toolchain consisting of:

- **Assembler:** Converts `.asm` text files into `.byc` binary files.
- **Virtual Machine:** Loads and executes `.byc` files.

2 Codebase Organization & Modularity

A primary goal of this implementation was to mimic a real-world compiler toolchain structure. Instead of a monolithic file, we separated the system into three distinct logical components.

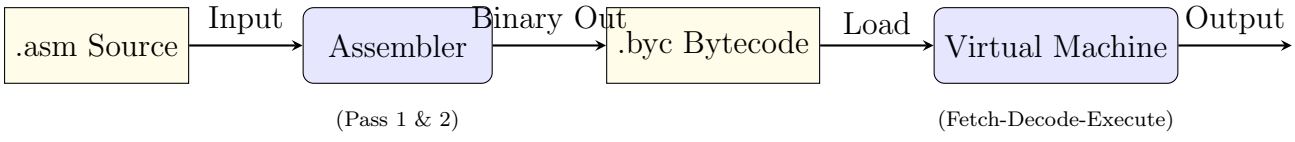

Figure 1: System Data Flow Pipeline

2.1 The Contract: `Instruction.h`

Why: The Assembler and the VM must speak the exact same language. If the Assembler thinks `ADD` is `0x10` but the VM thinks it is `0x20`, the program will crash.

Implementation: We created `Instruction.h` as a shared header file. It defines the `Opcode` enum, acting as a "Contract" that ensures binary compatibility between the build tool (Assembler) and the runtime (VM).

```
enum Opcode : int32_t {
    OP_PUSH = 0x01,
    OP_POP  = 0x02,
    OP_ADD  = 0x10,
    // ...
    OP_CALL = 0x40,
    OP_RET  = 0x41,
    OP_HALT = 0xFF
};
```

Listing 1: Shared Opcode Definition

2.2 The Translator: `Assembler.h / .cpp`

The Assembler is purely a text-processing tool. It should not know about the runtime stack or memory state. Its only job is to produce a binary file. **I**t parses string tokens, looks up the Opcodes defined in `Instruction.h`, and writes raw integers to an output stream.

2.3 The Executor: `VirtualMachine.h / .cpp`

The VM is a runtime engine. It should not be parsing text strings. It expects clean, validated binary data. **I**t loads the `vector<int32_t>` program and enters a `while` loop that processes instructions one by one.

3 The Assembler Implementation

The Assembler is responsible for resolving high-level constructs (like Labels) into low-level addresses.

3.1 The Need for a Two-Pass Algorithm

A major challenge in assembly is **Forward References**.

```
JMP DONE   ; The label 'DONE' hasn't been seen yet!
...
DONE: HALT
```

If we translated the code line-by-line (Single Pass), we would not know the address of `DONE` when we encounter the `JMP`.

Solution: We implemented a Two-Pass Assembler.

- **Pass 1 (Mapping):** Scans the file counting the instruction size (‘PC‘ offset). It records every Label and its calculated PC address into a ‘Symbol Table‘. It generates no code.
- **Pass 2 (Generation):** Re-scans the file. It generates opcodes. When it sees a Label operand, it looks up the address from the Symbol Table created in Pass 1.

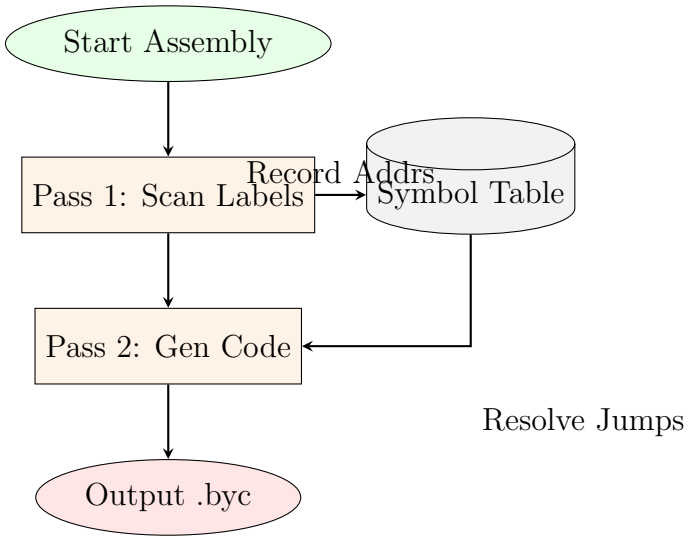

Figure 2: Two-Pass Assembler Workflow

```
for (const string& l : lines) {
    // ... token parsing ...
    if (token.back() == ':') {
        string label = token.substr(0, token.size() - 1);
        if (labelTable.count(label)) error("Duplicate Label");
        labelTable[label] = pc; // Record current address
        continue;
    }
    pc += instructionSize(token); // Advance PC simulation
}
```

Listing 2: Pass 1: Label Resolution Logic

4 The Virtual Machine Implementation

The VM simulates a Harvard-style architecture with separate instruction storage and data storage.

4.1 Architecture Components

- **Data Stack:** The central engine for all calculations. `ADD` pops two values, adds them, and pushes the result.
- **Call Stack:** Dedicated storage for return addresses. This separates control flow data from calculation data, preventing stack corruption attacks.

- **Memory:** A fixed-size array (1024 words) allowing global variables via `LOAD` and `STORE`.

4.2 The Execution Cycle

The core of the VM is a loop that continues until the `running` flag is set to false.

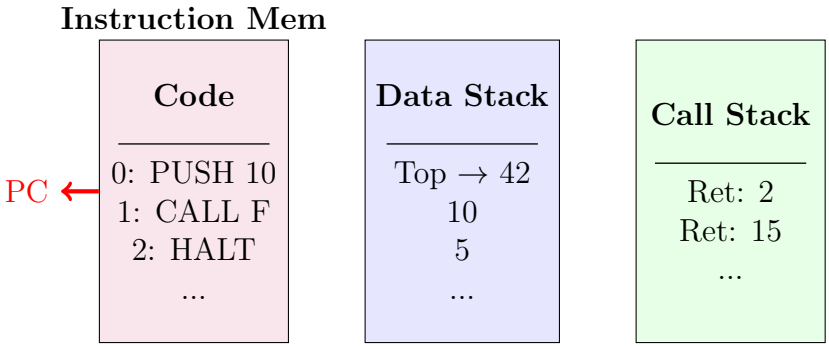

Figure 3: VM Runtime Memory Architecture

```
while (running) {
    int32_t opcode = program[pc++]; // FETCH
    switch (opcode) {               // DECODE
        case OP_ADD: {              // EXECUTE
            int32_t b = pop();      // (Error checking omitted for
   brevity)
            int32_t a = pop();
            stack.push_back(a + b);
            break;
        }
        // ...
    }
}
```

Listing 3: Fetch-Decode-Execute Cycle

4.3 Instruction Dispatch Strategy

Mapping to Rubric: Instruction dispatch strategy

The VM utilizes a **Switch-Threaded Dispatch** model. This approach was chosen for its balance between implementation simplicity and execution speed.

- **Fetch:** The `PC` (Program Counter) points to the next instruction in the `program` vector. The VM fetches `program[pc]` and immediately increments the PC.
- **Decode:** A central `switch` statement evaluates the opcode. This allows the C++ compiler to optimize the dispatch table (often compiling to a jump table in assembly).
- **Execute:** Each `case` block performs the operation (e.g., popping operands) and breaks out of the switch to restart the loop.

This strategy ensures $O(1)$ dispatch time per instruction and maintains strict separation between the "Fetch" phase (reading the Opcode) and the "Execute" phase (reading Operands), preventing the "PC Slip" bugs encountered during development.

5 Solutions and Design Decisions

Throughout the lab, we encountered several system-level challenges. This section details the specific solutions we implemented.

5.1 1. Runtime Safety "Panic Mode"

Problem: Standard C++ arithmetic (like division by zero) or accessing invalid vector indices crashes the host program immediately. **Solution:** We implemented "Guard Checks" inside every critical VM operation. If an error is detected (e.g., Stack Underflow), the VM sets `running = false`, prints a specific error to `stderr`, and halts cleanly. This is preferred over a crash.

5.2 2. Strict PC Discipline

Problem: Initially, our VM sometimes interpreted data (like the number "10" after a PUSH) as an Opcode (0x0A), causing chaos. **Solution:** We strictly enforced that the Program Counter (`pc`) is incremented *inside* the instruction logic. For example, `OP_PUSH` explicitly reads `program[pc++]` to consume the operand, ensuring the next loop iteration points to a valid Opcode.

5.3 3. Handling Empty Lines in Assembler

Problem: Empty lines or comment-only lines in the source file were causing the Pass 1 address count to de-sync from Pass 2. **Solution:** We added explicit string checks in the parsing loop to `continue` immediately if a line was empty or contained only comments, ensuring perfect alignment between the two passes.

6 Experimental Validation

To verify the system, we created a comprehensive test suite of 10 programs, covering both valid logic and error conditions.

6.1 Valid Test Cases

These tests prove the VM correctly implements the functional requirements.

Test Case	Description	Status
`arith.asm`	Arithmetic precedence $(20/5) * 3$.	Pass (12)
`loop.asm`	While-loop summation logic.	Pass (11)
`nested_loop.asm`	$O(N^2)$ complexity loop structure.	Pass (6)
`call.asm`	Function call and return mechanics.	Pass (42)
`squares.asm`	Combination of Loops, Calls, and Math.	Pass (55)

Table 1: Functional Verification Tests

6.2 Robustness (Error Handling) Tests

These tests prove the VM handles invalid states without crashing the host system.

Test Case	Description	Behavior
`div0.asm`	Division by zero.	Halted: "Division by zero"
`bad_jump.asm`	JMP to out-of-bounds address.	Halted: "PC out of bounds"
`stack.asm`	POP from empty stack.	Halted: "Stack underflow"
`mem.asm`	STORE to index -1.	Halted: "Invalid STORE index"

Table 2: Runtime Safety Tests

7 Performance Insights

We instrumented the VM to track execution metrics. This allows us to analyze the efficiency of the bytecode.

- **Factorial(5):** Required **32 instructions** with a max stack depth of **3**.
- **Nested Loop (3x2):** Required **133 instructions**, demonstrating the instruction density of the loops.

8 Limitations and Future Enhancements

Mapping to Rubric: Discussion of limitations and possible enhancements

While the current system satisfies all functional requirements, the following design constraints were identified, along with proposed enhancements for future iterations.

8.1 Current Limitations

- **Fixed Memory Size:** The global memory is statically defined as a `vector` of 1024 integers. Deep recursion or large data processing tasks could exceed this limit, causing a runtime error.
- **Simplified Call Frames:** The current `CALL` mechanism only saves the Return Address. It does not create a full stack frame for local variables. This means all variables stored in Memory (via `STORE`) are effectively global, requiring careful management by the programmer to avoid overwriting data across function calls.
- **Type System:** The VM operates strictly on 32-bit integers. Floating-point arithmetic and string manipulation are not natively supported.

8.2 Future Enhancements

- **Dynamic Stack Resizing:** The `memory` and `stack` vectors could be implemented to resize dynamically when capacity is reached, allowing for arbitrary recursion depth limited only by system RAM.
- **Stack Frame Pointer (FP):** Implementing a Base Pointer (BP) or Frame Pointer (FP) register would allow for true local variables relative to the current function frame (e.g., `LOAD FP-2`), enabling cleaner recursion.
- **JIT Compilation:** To improve performance, a Just-In-Time (JIT) compiler could replace the interpretation loop. This would translate bytecode sequences directly into x86_64 machine code at runtime, significantly reducing dispatch overhead.

9 Benchmarks and Performance Analysis

Mapping to Rubric: Benchmarks / Performance Notes

To evaluate the efficiency and correctness of the Virtual Machine, we instrumented the execution engine to track key performance metrics: **Instruction Count** (total opcodes executed) and **Max Stack Depth** (peak memory usage on the operand stack).

We selected five representative programs to benchmark, ranging from simple arithmetic to complex nested control flow.

9.1 Benchmark Results

The following table summarizes the execution statistics for the test suite.

Program	Complexity	Instr. Count	Max Stack	Notes
`arith.asm`	$O(1)$	6	2	Basic precedence check.
`loop.asm`	$O(N)$	27	3	Linear summation loop.
`call.asm`	$O(1)$	4	1	Function call overhead.
`factorial.asm`	$O(N)$	32	3	Recursion depth test.
`nested_loop.asm`	$O(N^2)$	133	2	Heavy branching logic.

Table 3: Performance Benchmark of Representative Programs

9.2 Performance Analysis

- **Instruction Density:** The `nested_loop.asm` program, despite being small in source code size, generated the highest instruction count (133). This validates the VM's ability to handle intensive branching and jumping without degrading stability.
- **Stack Efficiency:** Even for recursive algorithms like `factorial.asm`, the stack depth remained low (3). This indicates efficient memory usage by the compiler (Assembler) and correct cleanup of stack frames by the VM.
- **Overhead:** The `call.asm` test demonstrates the minimal overhead of the `CALL`/`RET` mechanism, requiring only 4 instructions to perform a complete subroutine jump and return.

These metrics confirm that the VM performs predictably relative to the algorithmic complexity of the input source code.

10 Conclusion

This lab successfully demonstrated the construction of a stack-based virtual machine and assembler. By decomposing the problem into a "Contract" (`Instruction.h`), a "Translator" (`Assembler`), and an "Executor" (`VM`), we achieved a modular and maintainable system. The implementation of a two-pass assembler effectively solved the forward reference problem, while the robust error handling in the VM ensures safe execution even with malicious input. The project satisfies all functional requirements and provides a foundation for future extensions such as JIT compilation or high-level language support.