Lab 4: Stack-Based Bytecode Virtual Machine

K RAJ KUMAR
Roll No:*2025MCS2744*

1 Introduction

This assignment involves the design and implementation of a stack-based bytecode virtual machine (VM) capable of executing a defined instruction set. The VM supports arithmetic operations, control flow, function calls, and memory access. In addition, a two-pass assembler is implemented to translate human-readable assembly programs into executable bytecode.

The objective of this lab is to understand low-level execution models, instruction dispatch, and runtime system design, similar in spirit to virtual machines such as the JVM or Python VM, but in a simplified and controlled setting.

—

2 System Architecture

The system consists of two main components:

- A **Bytecode Virtual Machine** that executes compiled bytecode.
- An **Assembler** that converts assembly language programs into bytecode.

These components are implemented as separate executables to maintain a clean separation of concerns.

2.1 Virtual Machine Components

The VM maintains the following runtime structures:

- **Program Counter (PC):** Tracks the current byte offset in the bytecode.
- **Operand Stack:** Used for arithmetic, logical operations, and intermediate values.
- **Return Stack:** Stores return addresses for function calls.
- **Memory Array:** Global memory accessed via LOAD and STORE instructions.
- **Bytecode Array:** Holds the loaded program instructions.

The VM follows a classical *fetch–decode–execute* cycle until a HALT instruction is encountered or an error occurs.

—

3 Instruction Set Design

The VM implements a stack-based instruction set as specified in the lab handout.

3.1 Data Movement and Stack Management

- `PUSH val`: Pushes a 32-bit integer onto the stack.
- `POP`: Removes the top element of the stack.
- `DUP`: Duplicates the top element.
- `HALT`: Terminates execution.

3.2 Arithmetic and Logical Instructions

- `ADD`, `SUB`, `MUL`, `DIV`: Perform arithmetic using the top two stack values.
- `CMP`: Pushes 1 if the second-top value is less than the top value, otherwise 0.

All arithmetic instructions enforce stack underflow checks, and division by zero is handled explicitly.

3.3 Control Flow Instructions

- `JMP addr`: Unconditional jump to a byte address.
- `JZ addr`: Jump if the popped condition value is zero.
- `JNZ addr`: Jump if the popped condition value is non-zero.

Jump targets are absolute byte offsets in the bytecode.

3.4 Memory and Function Calls

- `LOAD idx`, `STORE idx`: Access global memory.
- `CALL addr`: Saves return address and jumps to a function.
- `RET`: Restores the return address from the return stack.

A separate return stack is used to avoid interference with operand data.

—

4 Instruction Dispatch and Execution

Instruction dispatch is implemented using a `switch-case` construct on the opcode. Instructions are either single-byte or multi-byte (opcode + 32-bit operand). The program counter is manually updated depending on instruction size.

Safety checks include:

- Stack underflow and overflow detection
- Invalid opcode detection
- Invalid memory access
- Invalid jump or call addresses

Execution is deterministic and terminates cleanly on HALT or runtime error.

—

5 Assembler Design

The assembler translates assembly programs into bytecode using a **two-pass design**.

5.1 Pass 1: Label Resolution

In the first pass:

- The source file is scanned line by line.
- Labels are recorded with their corresponding byte offsets.
- Instruction sizes are used to compute addresses accurately.

5.2 Pass 2: Bytecode Generation

In the second pass:

- Instructions are translated into opcodes.
- Label references are replaced with resolved byte addresses.
- Operands are written in little-endian format.

This design ensures correct handling of forward and backward jumps.

—

6 Testing and Validation

Multiple test programs were written to validate each VM feature:

- Stack manipulation tests
- Arithmetic expression evaluation
- Conditional branching and loops
- Memory load/store operations
- Function calls and returns

- Non-trivial programs such as iterative loops and formula evaluation

All tests were assembled using the custom assembler and executed successfully on the VM.

—

7 Limitations

The current VM has the following limitations:

- Only integer data type is supported.
- No local variables or stack frames beyond return addresses.
- No garbage collection or dynamic memory allocation.
- No debugging or tracing facilities.

—

8 Possible Enhancements

Potential future improvements include:

- Support for local variables and stack frames.
- Additional data types and instructions.
- A standard library of common routines.
- Bytecode verification and debugging support.
- Just-In-Time (JIT) compilation for performance.

—

9 Conclusion

This lab provided hands-on experience in building a complete execution environment, from assembly translation to runtime execution. The project demonstrates fundamental concepts in virtual machine design, instruction dispatch, and low-level program execution, reinforcing core systems programming principles.

—