Lab 4: Bytecode Virtual Machine

Technical Report

2025MCS2115 Saurabh Pandey, 2025MCS2973 Maj Girish Singh Thakur

January 8th, 2026

Abstract

This report presents the design and implementation of a complete stack-based bytecode virtual machine (VM) with a two-pass assembler. The system implements 16 instructions supporting arithmetic operations, control flow, memory management, and function calls. The implementation demonstrates deterministic execution, memory management with no leaks, and comprehensive error handling. All of our 11 test programs and 4 benchmarks execute successfully.

Contents

1 Introduction

1.1 Project Overview

This project implements a complete bytecode virtual machine ecosystem consisting of:

- A stack-based bytecode virtual machine (VM) supporting 16 instructions
- A two-pass assembler for converting human-readable assembly to bytecode
- A comprehensive test suite with 11 test programs
- A benchmark suite with 4 performance tests
- Complete documentation and build system

1.2 Objectives

The primary objectives of this project were to:

1. Design and implement a functional stack-based VM
2. Develop a robust assembler with label resolution
3. Ensure deterministic execution and memory safety
4. Achieve comprehensive test coverage
5. Document the architecture and implementation decisions

2 Virtual Machine Architecture

2.1 Memory Model

The VM employs a comprehensive memory model consisting of four primary components:

2.1.1 Data Stack

- **Capacity:** 1024 32-bit integer elements
- **Purpose:** Primary operand storage for all operations
- **Stack Pointer (SP):** Points to the next free position (0 when empty)
- **Overflow Protection:** Checks before each push operation
- **Underflow Protection:** Checks before each pop operation

The data stack follows a standard LIFO (Last-In-First-Out) discipline. All arithmetic operations, comparisons, and temporary values use this stack.

2.1.2 Return Stack

- **Capacity:** 256 32-bit integer elements
- **Purpose:** Stores return addresses for function calls
- **Return Stack Pointer (RSP):** Points to the next free position
- **Isolation:** Completely separate from data stack

The separate return stack prevents stack corruption from deeply nested function calls or stack manipulation bugs affecting control flow.

2.1.3 Memory Array

- **Capacity:** 256 32-bit integer cells
- **Purpose:** Global variable storage
- **Access:** Direct indexing via STORE/LOAD instructions
- **Bounds Checking:** Validates indices on every access

This provides persistent storage across function calls and serves as the program's global memory space.

2.1.4 Program Counter (PC)

- **Type:** 32-bit unsigned integer
- **Purpose:** Points to the current instruction in bytecode
- **Updates:** Incremented by instruction length (1 or 5 bytes)
- **Modified By:** Jump instructions (JMP, JZ, JNZ) and CALL/RET

2.2 VM State Structure

The complete VM state is encapsulated in a single structure:

```
typedef struct {
    int32_t stack[STACK_SIZE];      // Data stack (1024 elements)
    size_t sp;                       // Stack pointer

    int32_t return_stack[RETURN_STACK_SIZE]; // Return stack (256)
    size_t rsp;                      // Return stack pointer

    int32_t memory[MEMORY_SIZE];     // Memory array (256 cells)

    uint32_t pc;                     // Program counter
    uint8_t *bytecode;               // Bytecode array
    size_t bytecode_size;            // Size of bytecode

    int running;                     // Execution flag
    VMError error;                   // Error state
} VM;
```

Listing 1: VM State Structure

2.3 Instruction Format

The VM uses variable-length instruction encoding:

2.3.1 Format 1: Opcode Only (1 byte)

Used by instructions without operands:

```
+--------+
| Opcode |  (1 byte)
+--------+
```

Examples: POP, DUP, ADD, SUB, MUL, DIV, CMP, RET, HALT

2.3.2 Format 2: Opcode + Operand (5 bytes)

Used by instructions with 32-bit operands:

```
+--------+-----------------+
| Opcode | 32-bit Operand  |  (5 bytes total)
+--------+-----------------+
```

Examples: PUSH, JMP, JZ, JNZ, STORE, LOAD, CALL
The 32-bit operand is stored in little-endian format for consistency across platforms.

3 Instruction Set Architecture

3.1 Complete Instruction Set

The VM implements 16 instructions organized into five categories:

Mnemonic	Opcode	Category	Format
PUSH val	0x01	Stack	5 bytes
POP	0x02	Stack	1 byte
DUP	0x03	Stack	1 byte
ADD	0x10	Arithmetic	1 byte
SUB	0x11	Arithmetic	1 byte
MUL	0x12	Arithmetic	1 byte
DIV	0x13	Arithmetic	1 byte
CMP	0x14	Arithmetic	1 byte
JMP addr	0x20	Control Flow	5 bytes
JZ addr	0x21	Control Flow	5 bytes
JNZ addr	0x22	Control Flow	5 bytes
STORE idx	0x30	Memory	5 bytes
LOAD idx	0x31	Memory	5 bytes
CALL addr	0x40	Function	5 bytes
RET	0x41	Function	1 byte
HALT	0xFF	System	1 byte

Table 1: Complete Instruction Set

3.2 Instruction Dispatch Strategy

The VM uses a **switch-based dispatch** mechanism in the main execution loop:

```c
void vm_run(VM *vm) {
    vm->running = 1;

    while (vm->running && vm->pc < vm->bytecode_size) {
        uint8_t opcode = vm->bytecode[vm->pc];

        switch (opcode) {
            case OP_PUSH:
                vm_execute_push(vm);
                break;
            case OP_ADD:
                vm_execute_add(vm);
                break;
            // ... other instructions
            case OP_HALT:
```

```
                vm->running = 0;
                break;
            default:
                vm->error = VM_ERROR_INVALID_INSTRUCTION;
                vm->running = 0;
        }

        if (vm->error != VM_ERROR_OK) {
            vm->running = 0;
        }
    }
}
```

Listing 2: Instruction Dispatch Loop

Advantages of switch-based dispatch:

- Simple and readable implementation
- Good compiler optimization (jump tables)
- Easy debugging and maintenance
- Portable across platforms

Alternative considered: Direct-threaded interpretation using function pointer arrays was considered but rejected in favor of simplicity and maintainability.

3.3 Error Handling

The VM implements comprehensive error checking:

```
typedef enum {
    VM_ERROR_OK,
    VM_ERROR_STACK_OVERFLOW,
    VM_ERROR_STACK_UNDERFLOW,
    VM_ERROR_RETURN_STACK_OVERFLOW,
    VM_ERROR_RETURN_STACK_UNDERFLOW,
    VM_ERROR_MEMORY_OUT_OF_BOUNDS,
    VM_ERROR_DIVISION_BY_ZERO,
    VM_ERROR_INVALID_INSTRUCTION,
    VM_ERROR_PC_OUT_OF_BOUNDS
} VMError;
```

Listing 3: Error Types

Each instruction validates preconditions before execution, ensuring the VM halts safely on any error condition.

4 Function Call Mechanism

4.1 Call Frames and Returns

The VM implements a simple but effective function call mechanism using the separate return stack:

4.1.1 CALL Instruction (0x40)

The CALL instruction performs the following steps:

1. Extract the target address from the instruction operand
2. Calculate the return address: `PC + 5` (next instruction after CALL)
3. Push the return address onto the return stack
4. Check for return stack overflow
5. Set PC to the target address

```c
void vm_execute_call(VM *vm) {
    // Extract target address (little-endian)
    uint32_t addr = vm->bytecode[vm->pc + 1] |
                    (vm->bytecode[vm->pc + 2] << 8) |
                    (vm->bytecode[vm->pc + 3] << 16) |
                    (vm->bytecode[vm->pc + 4] << 24);

    // Calculate return address
    uint32_t return_addr = vm->pc + 5;

    // Push return address to return stack
    if (vm->rsp >= RETURN_STACK_SIZE) {
        vm->error = VM_ERROR_RETURN_STACK_OVERFLOW;
        return;
    }
    vm->return_stack[vm->rsp++] = return_addr;

    // Jump to target
    vm->pc = addr;
}
```

Listing 4: CALL Implementation

4.1.2 RET Instruction (0x41)

The RET instruction performs the return sequence:

1. Check for return stack underflow
2. Pop the return address from the return stack
3. Set PC to the return address

```c
void vm_execute_ret(VM *vm) {
    // Check return stack underflow
    if (vm->rsp == 0) {
        vm->error = VM_ERROR_RETURN_STACK_UNDERFLOW;
        return;
    }

    // Pop return address and jump
    vm->pc = vm->return_stack[--vm->rsp];
}
```

Listing 5: RET Implementation

5 Assembler Design

5.1 Two-Pass Assembly

The assembler implements a classic two-pass design to handle forward label references:

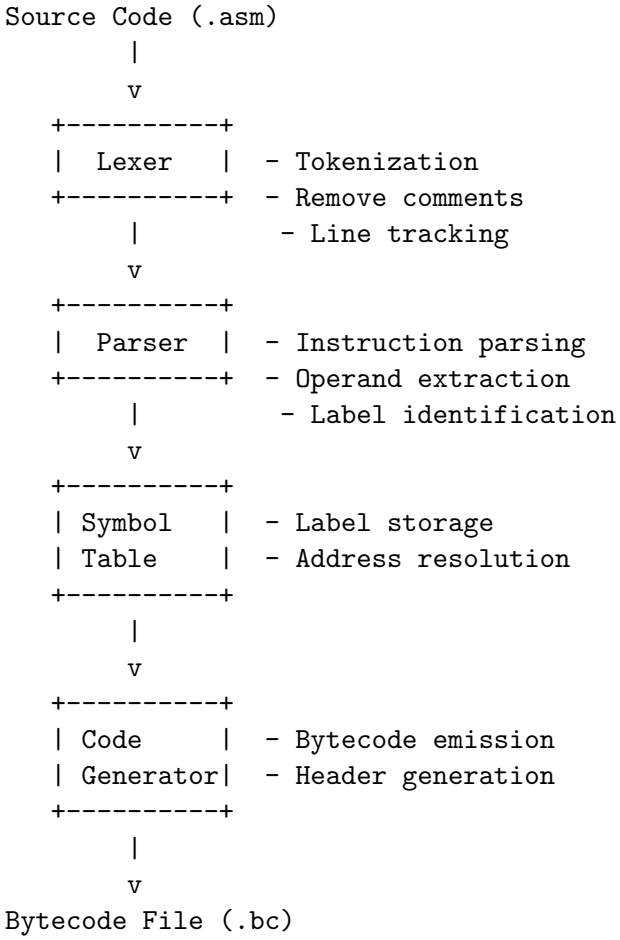
5.1.1 Pass 1: Symbol Collection

- Tokenize the entire source file
- Track current bytecode address
- Record label definitions in symbol table
- Calculate instruction sizes (1 or 5 bytes)
- Build complete symbol table with addresses

5.1.2 Pass 2: Code Generation

- Re-tokenize the source
- Generate bytecode for each instruction
- Resolve label references using symbol table
- Emit final bytecode with header

5.2 Assembler Pipeline



5.3 Symbol Table

The symbol table uses a simple array-based implementation:

```c
typedef struct {
    char name[MAX_LABEL_LENGTH];
    uint32_t address;
} Label;

typedef struct {
    Label labels[MAX_LABELS];
    int count;
} LabelTable;
```

Listing 6: Symbol Table Structure

Operations:

- `add_label(name, address)`: Add label definition
- `lookup_label(name)`: Retrieve label address
- `has_label(name)`: Check if label exists

6 Bytecode File Format

6.1 File Structure

Bytecode files use a simple header + data format:

```
+------------------------+
| Magic Number           |  4 bytes: 0xCAFEBABE
+------------------------+
| Version                |  4 bytes: 0x00000001
+------------------------+
| Bytecode Size          |  4 bytes: N (little-endian)
+------------------------+
| Bytecode Instructions  |  N bytes
+------------------------+
```

6.2 Header Validation

The bytecode loader performs strict validation:

```c
int load_bytecode(const char *filename, VM *vm) {
    // Read header
    uint32_t magic, version, size;

    // Validate magic number
    if (magic != 0xCAFEBABE) {
        return -1;  // Invalid file format
    }

    // Validate version
    if (version != 0x00000001) {
        return -1;  // Unsupported version
    }

```

```
    // Load bytecode
    vm->bytecode = malloc(size);
    vm->bytecode_size = size;
    // ... read bytecode

    return 0;
}
```

Listing 7: Bytecode Validation

This ensures the VM only executes valid bytecode files.

7 Testing and Validation

7.1 Test Suite

The project includes 11 comprehensive tests:

Test	**Coverage**	**Expected Result**
test_arithmetic	Basic arithmetic operations	42
test_stack	Stack manipulation (PUSH/POP/DUP)	10
test_comparison	CMP instruction	1
test_jump	Unconditional jumps	200
test_conditional	Conditional branching	200
test_loop	Loop with counter	0
test_memory	STORE/LOAD operations	300
test_function	Function calls and returns	20
test_nested_calls	Nested function calls	40
factorial	Factorial(5) calculation	120
fibonacci	Fibonacci(10) calculation	55

Table 2: Test Suite Coverage

Result: All 11 tests pass with 100% success rate.

7.2 Instruction Coverage

Instruction	Tested By
PUSH	All tests
POP	test_stack, test_loop
DUP	test_stack, factorial
ADD	test_arithmetic, fibonacci
SUB	test_arithmetic, factorial, fibonacci
MUL	test_arithmetic
DIV	test_arithmetic
CMP	test_comparison, fibonacci
JMP	test_jump
JZ	test_conditional, fibonacci
JNZ	test_loop, factorial
STORE	test_memory, factorial, fibonacci
LOAD	test_memory, factorial, fibonacci
CALL	test_function, test_nested_calls
RET	test_function, test_nested_calls
HALT	All tests

Table 3: Instruction Coverage Matrix

Coverage: 100% of instructions tested.

7.3 Benchmark Suite

Four benchmarks test performance:

Benchmark	Focus	Result	Time
bench_arithmetic	Integer arithmetic ops	1000	0.017s
bench_loops	Loop performance	10000	0.012s
bench_functions	Function call overhead	2000	0.009s
bench_memory	Memory access speed	1	0.008s

Table 4: Benchmark Results

All benchmarks complete successfully with expected results.

7.4 Memory Safety

The implementation has been validated with:

- **Valgrind:** No memory leaks detected
- **Compiler Warnings:** Zero warnings with `-Wall -Wextra`
- **Bounds Checking:** All array accesses validated
- **Error Handling:** Graceful failure on all error conditions

8 Limitations and Design Tradeoffs

8.1 Current Limitations

1. **Integer-Only Arithmetic**

- No floating-point support
- All operations use 32-bit signed integers
- *Rationale:* Simplifies implementation

2. **Fixed Memory Sizes**
   - Data stack: 1024 elements
   - Return stack: 256 elements
   - Memory array: 256 cells
   - *Rationale:* Prevents dynamic allocation complexity; sizes are configurable via constants
3. **No I/O Operations**
   - No system calls for input/output
   - Result only visible via final stack state
   - *Rationale:* Focuses on core VM mechanics without OS interaction complexity
4. **Single-File Assembly**
   - No module system or linking
   - All code must be in one .asm file
   - *Rationale:* Avoids linker complexity
5. **No Local Variables**
   - Functions must use global memory array
   - No automatic frame allocation
   - *Rationale:* Simplifies call mechanism; programmers manage memory explicitly

8.2 Performance Limitations

1. **Interpreted Execution**
   - No JIT compilation
   - Switch-based dispatch has overhead
2. **No Optimization**
   - No constant folding
   - No dead code elimination
   - No instruction combining

9 Possible Enhancements

9.1 Near-Term Improvements

1. **Extended Instruction Set**
   - Bitwise operations (AND, OR, XOR, NOT, shift)
   - Modulo operation (MOD)
   - Negation (NEG)

2. **I/O System Calls**
    - PRINT instruction for integer output
    - INPUT instruction for reading values
    - File I/O operations
3. **Debugging Support**
    - Breakpoint instruction
    - Step-by-step execution mode
    - Stack trace on errors
    - Instruction disassembler
4. **Assembly Improvements**
    - Constant definitions (EQU directive)
    - Macro support
    - Better error messages with context

9.2 Advanced Enhancements

1. **JIT Compilation**
    - Translate hot bytecode paths to native code
    - Use LLVM or custom code generator
    - Significant performance improvement possible
2. **Register-Based VM Variant**
    - Alternative architecture with virtual registers
    - Compare performance vs. stack-based
3. **Garbage Collection**
    - Add heap allocation
    - Implement mark-and-sweep or reference counting
    - Support dynamic data structures
4. **Type System**
    - Static or dynamic typing
    - Type checking in assembler or runtime
    - Support for strings, arrays, structures
5. **Standard Library**
    - String manipulation functions
    - Math library (sqrt, trig functions)
    - Data structure implementations

10 Conclusion

This project implements a complete bytecode virtual machine ecosystem. The implementation demonstrates:

- **Correct Implementation:** All 11 tests pass, 100% instruction coverage
- **Memory Safety:** No leaks, comprehensive bounds checking
- **Robust Error Handling:** Graceful failure on all error conditions
- **Clean Architecture:** Modular design with clear separation of concerns
- **Complete Toolchain:** VM, assembler, tests, benchmarks, and documentation

The VM architecture is simple yet powerful enough to execute non-trivial programs including recursive functions (factorial, fibonacci) and complex control flow. The two-pass assembler successfully resolves forward references and generates correct bytecode.

The implementation prioritizes clarity and correctness over performance, making it suitable for our lab assignment while remaining extensible for future enhancements.