TECHNICAL REPORT

Bytecode Virtual Machine (BVM)

GitHub: Bytecode_Virtual_Machine

Assignment: Lab 4
Date: 07-01-2026

Team: Abhinav Gupta (2025MCS2089)
Pratik Chaudhari (2025MCS2096)

Abstract

This report documents the design, implementation, and operational logic of the **Bytecode Virtual Machine (BVM)**, a custom process virtual machine developed for COP7001 lab. BVM is a stack-based machine that executes a bytecode instruction set, simulating the fundamental operations of a CPU. The project includes a runtime environment (the VM), a translation tool (the Assembler), and a suite of test programs. This document details of memory architecture, the fetch-decode-execute cycle implementation, the instruction set architecture (ISA), and the system's error-handling mechanisms.

Virtual Machines (VMs) provide an abstraction layer over physical hardware, allowing software to run independently of the underlying platform. BVM is designed as a **stack machine,** where operands are strictly popped from a data stack, processed, and pushed back. This results in a compact bytecode density, as instructions often require zero operands (e.g., ADD, MUL).

System Design & Architecture

The BVM simulates a simplified computer architecture defined in src/vm/bvm.h. It follows a model where the executable code (bytecode) is stored in a separate memory space from the runtime data.

Memory Layout: The VM manages three distinct memory regions, allocated on the host machine's stack during initialization:

1. **Program Memory (program):**
   - **Type:** unsigned char*
   - **Description:** A read-only buffer containing the bytecode loaded from a .bin file.
   - **Size:** Fixed at 1024 bytes (CODE_SIZE).
   - **Role:** Stores the sequence of opcodes and immediate values generated by the assembler.
2. **Operand Stack (stack[]):**
   - **Type:** int array (Size: 1024)
   - **Description:** The central data structure for all computation.
   - **Behavior:** Last-In, First-Out (LIFO).
   - **Role:** Arithmetic instructions (like ADD, SUB) act on the top two elements of this stack.
3. **Data Memory (memory[]):**
   - **Type:** int array (Size: 256)

    - **Description:** A Random Access Memory (RAM) simulation for global variables.
    - **Role:** Allows data to persist beyond the immediate scope of stack operations. It is accessed via directly addressing indices (0-255).
4. **Return Stack (ret_stack[]):**
    - **Type:** int array (Size: 256)
    - **Role:** Dedicated storage for control flow management. When a function is called, the return address (current PC offset) is pushed here, isolating control logic from data logic.

The VM state is maintained by a minimal set of registers simulated as class member variables:

- **Instruction Pointer (inst_ptr):** Equivalent to the Program Counter (PC). It points to the *next* byte to be executed in the program buffer.
- **Stack Pointer (st_ptr):** An integer index indicating the next free slot in the stack. st_ptr = 0 implies an empty stack.
- **Return Stack Pointer (rst_ptr):** An integer index tracking the depth of nested function calls.
- **Running Flag (running):** A boolean latch that keeps the main execution loop active until a HALT instruction or critical error occurs.

Instruction Set Architecture (ISA)

The BVM executes a custom 32-bit integer ISA. The opcodes are defined in src/commons.h and categorized into data movement, arithmetic, control flow, and system operations.

Supported Operations

Mnemonic	Opcode	Operand	Stack Transition	Description
PUSH	0x01	Integer	[] -> [val]	Pushes a 32-bit integer onto operand stack.
POP	0x02	None	[val] -> []	Discards the top element of the stack.
DUP	0x03	None	[a] -> [a, a]	Duplicates the top stack element.
ADD	0x10	None	[a, b] -> [a+b]	Pops two values, pushes their sum.
SUB	0x11	None	[a, b] -> [a-b]	Pops b then a, pushes a-b.
MUL	0x12	None	[a, b] -> [a*b]	Pops two values, pushes their product.
DIV	0x13	None	[a, b] -> [a/b]	Pops two values, pushes a/b. Traps on zero.
CMP	0x14	None	[a, b] -> [0/1]	Pushes 1 if a < b, else 0.
JMP	0x20	Address	No Change	Unconditional jump to bytecode address.
JZ	0x21	Address	[val] -> []	Pop val; if val == 0, jump to address.
JNZ	0x22	Address	[val] -> []	Pop val; if val != 0, jump to address.
STORE	0x30	Index	[val] -> []	Pops value into memory[Index].
LOAD	0x31	Index	[] -> [val]	Pushes value from memory[Index].
CALL	0x40	Address	No Change	Pushes PC+5 to return stack; jumps to addr.
RET	0x41	None	No Change	Pops address from return stack; restores PC.
HALT	0xFF	None	No Change	Stops the VM execution loop.

Virtual Machine Implementation

The core logic resides in src/vm/bvm.cpp. The VM::run() method acts as the CPU's control unit, executing a continuous **Fetch-Decode-Execute** cycle.

The Execution Cycle

1. **Fetch:** The byte currently pointed to by this->inst_ptr is dereferenced to retrieve the opcode.
2. **Decode:** The switch statement is used to interpret the opcode. This provides O(1) jump table efficiency in compiled code.
3. **Operand Extraction:** For instructions requiring an argument (like PUSH or JMP), the VM performs a direct memory read from the instruction stream. It casts the memory address immediately following the opcode to an int* and dereferences it:
   *int val = *(int *)(this->inst_ptr + 1);*
   This technique effectively reads the next 4 bytes as a single integer.

Arithmetic Logic Unit (ALU)

The arithmetic operations (ADD, SUB, MUL, DIV, CMP) directly manipulate the stack.

- **Logic:** They decrement the st_ptr to pop operands a and b.
- **Execution:** The operation is performed (e.g., int res = a + b;).
- **Write-back:** The result res is written to stack[st_ptr] and the pointer is incremented.
- **Safety:** Before any operation, check_stack(count) is called to ensure enough operands exist, preventing buffer underflows.

Control Flow Engine

The VM supports complex control flow necessary for completeness.

- **Branching:** JZ (Jump Zero) and JNZ (Jump Not Zero) inspect the top stack value. If the condition is met, the inst_ptr is forcefully updated to this->program + target, effectively "jumping" to a new code section.
- **Subroutines:** The CALL and RET instructions implement a hardware-supported call stack.
  - **CALL:** Calculates the return address (inst_ptr - program + 5) and pushes it to ret_stack.
  - **RET:** Retrieves this address and sets inst_ptr back to it.
  - **Significance:** This mechanism enables **recursion,** as validated by the factorial.asm test case which computes 5! using recursive calls.

Assembler Implementation

The Assembler (src/assembler/assembler.cpp) bridges the gap between human-readable code and the VM's binary format.

Parsing Logic

The assembler processes the input .asm file line-by-line.

1. **Tokenization:** It uses sscanf to parse lines into an instruction string (mnemonic) and an optional numeric argument.

2. **Comment Handling:** Lines starting with ; or empty lines are ignored, allowing for documented code.

Code Generation

For each valid line, the assembler:

1. **Opcode Mapping:** Compares the mnemonic string (e.g., "PUSH") against known commands to find the matching enum value (e.g., 0x01).
2. **Emission:**
   - **emit():** Writes the single-byte opcode to the binary buffer.
   - **emit_int():** If the instruction takes an operand, this helper function writes the integer into the buffer byte-by-byte. This ensures the binary format matches exactly what the VM expects to read.

Error Handling & Stability

The VM implements several safety checks to prevent crashes and undefined behavior:

1. Stack Overflow Protection:
   In PUSH and DUP, the VM checks if (st_ptr >= STACK_SIZE). If the stack limit (1024) is reached, it prints an error and halts execution, preventing memory corruption.
2. Stack Underflow Protection:
   The helper function check_stack(int count) is called before every POP or arithmetic operation. It ensures that the required number of operands exist on the stack before proceeding.
3. Division by Zero:
   The DIV instruction explicitly checks if the divisor is 0. If so, it prints "Error: Div by Zero" and halts, preventing a hardware exception on the host machine.
4. Unknown Opcodes:
   The default case in the main switch statement catches invalid bytes in the instruction stream, halting the VM gracefully.

Build System & Compilation

The project utilizes a makefile to automate the build process, defining specific targets for modular compilation:

- **make vm:** Compiles src/vm/bvm.cpp and src/vm/bvm_main.cpp into the vm executable.
- **make assembler:** Compiles the assembler logic into the assembler executable.
- **make benchmark:** Compiles the benchmarking tool which links against both VM and Assembler object files.
- **make run:** The make run target is a comprehensive automation pipeline that compiles the project dynamically, assembles and executes every test case found in the tests/ directory.

This modular build system ensures that changes in the VM core do not strictly require recompiling the assembler, and vice versa.