Technical Report: Bytecode Virtual Machine

1. VM Architecture

The virtual machine is designed as a stack-based architecture. This means that all arithmetic and logical operations retrieve their operands from a main `register_stack` and push their results back onto it. This simplifies the instruction format as operands do not need to be specified explicitly.

Components

- **Program Counter (PC):** An `unsigned long` that acts as an index into the `program_memory`, pointing to the next instruction to be executed.
- **Program Memory:** A `long` array that stores the bytecode loaded from a `.bin` file. Each instruction and operand occupies one `long` slot.
- **Register Stack:** A LIFO stack implemented as a `long` array. It is used for all operations, including arithmetic, function arguments, and local variables.
- **Call Stack:** A separate stack to store return addresses for function calls, enabling nested function calls.
- **Data Memory:** A general-purpose `long` array for storing and loading values using the `STORE` and `LOAD` instructions.

Instruction Dispatch

The VM uses a standard `while` loop that runs as long as the `HALT` instruction has not been executed. Inside the loop, it fetches the next `long` from `program_memory` at the `pc` address. This `long` is cast to an `Opcode` enum, and a `switch` statement dispatches to the correct logic for that instruction. This is a simple and direct dispatch method.

2. Assembler Design

The assembler is a crucial part of the toolchain, converting human-readable assembly language into the `long`-based bytecode that the VM executes. It is implemented using Flex (for lexical analysis) and Bison (for parsing).

Two-Pass Design

To handle forward references (i.e., using a label before it is defined), the assembler is designed as a **two-pass assembler**:

1. **First Pass:** The parser runs through the entire source file. Its only goal is to find all label definitions (e.g., `my_label:`). It stores each label and its address (the value of the program counter `pc` at that point) in a symbol table. Instructions that use labels as operands (like `JMP`, `JZ`, `JNZ`, `CALL`) are given a placeholder operand of `0`.

2. **Second Pass:** The file pointer and line counter are reset, and the parser runs again. This time, when it encounters an instruction that uses a label, it looks up the label in the symbol table to get its address. It then emits the correct `long` opcode and `long` address into the bytecode buffer.

This design allows for flexible use of labels for control flow without imposing declaration-before-use restrictions.

Grammar

The grammar is defined in `parser.y` and is line-oriented. It recognizes instructions (e.g., `PUSH 10`), label definitions (`loop_start:`), comments, and newlines. The grammar rules ensure that opcodes and operands are correctly parsed and passed to the `emit_long` function, which writes them to the bytecode buffer.

3. Call/Return Mechanism

The VM supports basic function calls and returns using the `CALL` and `RET` instructions.

- **`CALL addr`:** Pushes the current program counter (`pc`, which points to the instruction *after* the `CALL`) onto the `call_stack` and then unconditionally jumps to the specified `addr`.
- **RET:** Pops the return address from the `call_stack` back into the `pc`, resuming execution where the caller left off.

Calling Convention

We opted for a simple, manual "callee-cleans" calling convention.

- **Caller:** Pushes arguments onto the register stack before making a `CALL`.
- **Callee:** Is responsible for its own stack management. It must pop its arguments and any local variables it used, and then push its single return value (if any) before executing `RET`.

This approach avoids the complexity of implementing full stack frames with a frame pointer in the VM, trading automatic stack management for programmer diligence. It is a valid and efficient strategy for simpler VMs.

4. Limitations & Possible Enhancements

- **Error Handling:** The assembler and VM have basic error handling (e.g., "Label not found", "Stack Underflow"), but it could be made more robust with more specific error messages and recovery mechanisms.
- **Strict `long`-based format:** The current architecture uses `long`s for everything, including opcodes that only need a single byte. This makes the bytecode files larger than necessary. A more compact format could be used, where the VM reads operands of different sizes (1, 4, or 8 bytes) depending on the opcode.
- **No Local Scopes:** Without true call frames, there is no concept of local scopes. All values are on the global `register_stack` or in the global `data_memory`, which can make writing complex, modular code more difficult.

- **Assembler Syntax:** The assembler is quite strict (e.g., requiring labels on their own lines). It could be enhanced to be more flexible.

5. Testing Strategy

The project employs a multi-layered testing strategy to ensure correctness and robustness.

- **C++ Unit Tests (`make test`):** These tests, located in the `test/` directory, verify the core functionality of individual components like `Stack`, `Memory`, and `Opcode` handling in isolation.
- **Assembler Test Suite (`make -C Assembler test`):** The assembler has its own dedicated test suite. It assembles a series of `.asm` files and performs a binary `diff` against `.bin.expected` files to guarantee that the generated bytecode is exactly correct.
- **End-to-End Pipeline Tests (`make pipeline_test`):** This is the most comprehensive suite. It runs a series of assembly programs from the `pipeline_tests/` directory through the entire toolchain: assembly and then execution on the VM. The script automatically verifies the final state of the stack against expected values, confirming that the VM's instruction implementations are correct.
- **Benchmarks (`make benchmark`):** A suite of performance tests to measure the VM's execution speed on various tasks.

6. Benchmarks / Performance Analysis

To evaluate the performance of the VM, a suite of benchmarks was created and run. The benchmarks were executed on the same machine and the `real` time was recorded.

Results

Benchmark	Time (real)	Description
`simple_loop`	0.171s	A loop that executes 1,000,000 times with a few simple instructions per iteration.
`iterative_factorial`	0.002s	Calculates the factorial of 20 using an iterative loop.
`recursive_fibonacci`	0.001s	Calculates the 20th Fibonacci number using recursion.

Analysis

- **`simple_loop`:** This benchmark provides a baseline for the overhead of the main VM loop and simple instructions. The execution time is significant due to the high number of iterations (1,000,000), which highlights the cost of the instruction dispatch mechanism.
- **`iterative_factorial`:** This benchmark is very fast, as it involves a very small number of loop iterations (20). It demonstrates that for simple, short-running programs, the VM's performance is excellent.

- **`recursive_fibonacci`:** The recursive Fibonacci benchmark is also very fast. For `fib(20)`, the number of function calls is significant, but not large enough to incur a major performance penalty. This indicates that the `CALL` and `RET` instructions, while simple, are implemented efficiently. The performance of this benchmark would degrade significantly with larger input numbers, as expected from a recursive Fibonacci implementation.