Advanced Computing Systems Project: Virtual Machine

1. Introduction

The Virtual Machine project creates a basic 64-bit Virtual Machine (VM) written in C++. It is intended to illustrate essential computer architecture fundamentals such as a CPU, registers, memory, and the basic instruction set. The application includes an interactive Read-Eval-Print Loop (REPL) that allows users to run individual VM commands and see their impact on the system's state in real time. The virtual machine is a useful instructional tool for studying low-level computer ideas.

2. System Architecture

The virtual machine program is comprised with three primary components:

1) **OpCode Enum:** It defines the virtual machine's instruction set. Each instruction is encoded by a single byte (uint8_t), making the bytecode small and simple to process. The specified opcodes are **HALT, LOADI, ADD, SUB, MUL, DIV, PRINT, JUMP,** and **JUMP_EQ.**
2) **CPUState Struct:** The structure captures the CPU's current state. It includes an array of eight 64-bit registers (uint64_t registers[8]) and a Program Counter (uint64_t pc), which stores the memory location of the next instruction to be executed.
3) **VirtualMachine Class:** This is the primary class that controls the VM's activities. It includes the CPUState instance and a std::vector<uint8_t> that represents the VM's RAM. The class's main method, run(), performs the fetch-decode-execute cycle. Execute_single_instruction() is a distinct function that handles the interactive REPL.

3. Instruction Set Details

The virtual machine's instruction set comprises both data implementation and control flow instructions.

OpCode	Name	Format	Description
HALT	Halt Execution	**HALT**	Stops the VM's execution.
LOADI	Load Immediate	**LOADI R<dest>, <value>**	Loads a 64-bit literal value into a destination register.

ADD	Add Registers	**ADD R<dest>, R<src1>, R<src2>**	Adds the values of two source registers and stores the result in a destination register.
SUB	Subtract Registers	**SUB R<dest>, R<src1>, R<src2>**	Subtracts the value of the second source register from the first and stores the result.
MUL	Multiply Registers	**MUL R<dest>, R<src1>, R<src2>**	Multiplies the values of two source registers and stores the result.
DIV	Divide Registers	**DIV R<dest>, R<src1>, R<src2>**	Divides the values of the first source register by second and stores the result
PRINT	Print Register	**PRINT R<src>**	Prints the 64-bit value of source register to the console.
JUMP	Unconditional Jump	**JUMP <address>**	Sets the program counter to a new 64-bit memory address.
JUMP_EQ	Jump if Equal	**JUMP_EQ R<src>, <address>**	If the value in the source register is zero, sets the program counter to the new address.

4. Interactive REPL

The main() method opens an interactive REPL for the user. It repeatedly begs for a single instruction, which is subsequently sent to the **VirtualMachine::execute_single_instruction()** function. This approach enables for responsive and stateful interaction since the VM's registers and memory remain active between instructions. Error handling is implemented to detect erroneous instructions or operations, such as division by zero, and provide explicit feedback to the user. The REPL is designed to automatically handle input with or without commas, making it easier to use.

5. Implementation Details

Since the virtual machine's memory is a fixed-size **std::vector<uint8_t>**, this small model is adequate to demonstrate fundamentals concepts. Then the **execute_single_instruction** function tokenizes the user's input line using a **std::stringstream**. This enables flexible and robust parsing of the instruction and its operands. The **std::map<std::string, int>** function rapidly and reliably converts human-readable register names (e.g., "R0") to their numerical indices. The **memcpy** function reads multi-byte data directly from a byte-oriented memory vector, including 64-bit addresses and immediate values.

6. Future Improvements

Additional opcodes might be added to handle more complicated operations, such as bitwise operations (AND, OR, XOR), memory access (LOAD/STORE) and advanced conditional jumps. Then memory management unit will be used to prevent out-of-bounds memory access. A stack to handle function calls and returns will be created for allowing more complicated structures. Besides, the current assembler is rudimentary and it might be developed to include more complicated features like macros, multiple data kinds, and advanced directives.