Virtual CPU + VM

✱ High level overview :-

This project simulates a simple 16 bit CPU (Virtual Machine) in C++
- Load and execute instructions (like MOV, ADD, SUB, etc)
- Emulate memory and registers.
- Interpret the basic instruction set
- Execute small "programs" written as instructions

✱ It is made up of 5-parts :-

File	Purpose
main.cpp	the entry point :- runs the VM and test programs
RohitVM.hpp	Declares VM classes: CPU, Memory, Instruction, etc.
RohitVM.cpp	Implements the behaviour of the VM and how it exe
Rohitutils.hpp	~~Implements~~ Declares the helper function
Rohitutils.cpp	Implements those helper functions

✱ Short Summary of all 5 files :-

⇒ RohitVM.hpp – The Blueprint (classes & structures)

This file declares all core components of VM:

- enum class opcode :- list of all CPU instructions
- Struct instruction – represents of single instruction like `MOV R1,10`.
- class memory – simulates ram (memory space)
- class CPU – simulates CPU registers and program counters
- class VM – control the whole machine (fetch, decode, execute loop)

why it's needed:

– It defines the building blocks (registers, ram, instruction format, CPU logic) for your VM to function

⇒ RohitVM.cpp – The engine (logic implementation)

This file implements what the CPU actually does:

- Parses and fetches instruction
- Executes them using executeInstruction.
- Supports different instructions like MOV, ADD, SUB, JMP, HLT etc.

why it's needed?

– without this logic VM is just a shell. This is where the CPU 'thinks' and acts, handling real instruction behaviour.

⇒ Rohitutils.hpp / .cpp — Helper tools

These files:
- define and implements utility functions like split, tolowercase, etc.
- Mostly used for parsing instruction strings into usable data.

Why it's needed:
- Because our instruction input is string-based (MOV, R1, 10), you need helper functions to parse and interpret them correctly.

⇒ main.cpp – The launcher

This is the main file that:
- creates instruction sequence.
- Load them into the VM.
- starts execution.

why it's needed:
- It's the entry point of the application and shows how to use the VM you built. Also useful for testing different programs.

Detail Explanation of each file

1) RohitVM.hpp - "The blueprint header"

#pragma once ← ensures file only included once per compilation to avoid duplicate definition error

This file contain all the class, enum and struct declarations used to model your virtual CPU system. Think of this as the architecture or blueprint of your machine.

- **enum class opcode**

```
enum class opcode {
        MOV, ADD, SUB, MUL, DIV,
        JMP, CMP, JE, JNE, JG, JL, HLT, PRINT };
```

→ what it does :-

- Defines instruction types supported by your CPU.
- enum class makes the names scoped and type-safe (VS regular enum)

→ why needed :

You'll use this identify what kind of operation a given instruction is doing - like MOV (move value), ADD, HLT (Halt) etc.

- **Registers class**

it has General purpose registers like ax, bx, cx, dx

SP → stack pointer

ip → instruction pointer

flags → flag register

- uint16_t – unsigned 16 bit integer in C (defined by <stdint.h> and make sure 16 bit with of it
This simulates the CPU's internal memory. A real CPU has tiny storage locations called registers that are superfast.
- ax, bx, cx, dx → works like pockets to hold numbers for arithmetic
- sp (stack pointer): Points to the top of stack, which is temporary memory.
- ip (instruction ptr) : tells next instruction in memory is.
- flags: special bits that remember results of comparision (equal, greater, etc).

```
enum flag {
    equal = 0x08;
    greater 0x04,
    hisror 0x02;
    lower = 0x01 };
```

These are flag bits inside the flags register. They work like tiny switches: 0/1 (ON/OFF) that CPU uses to make decision like an if condition.

CPU class

This class wraps around the registers and gives functions to GET and SET flag values.

isEqual(), setEqual(True), setEqual(False)

- isEqual() → Check if equal flag is ON
- setEqual(True) → Turn ON equal flag
- setEqual(False) → Turn OFF equal flag

* It uses bit masking (a binary trick) to flip individual bits in the flags register

Memory class

It creates 64KB of ram

- std::vector<uint8_t> holds memory as bytes
- operator[]: allows you to access memory with syntax like
  memory[0x1234] = 42
- raw(): gives a pointer to the actual byte array, useful for copying or printing.

This simulates the computer's main memory - the place where code and data live.

Opcode Enum

```
enum class opcode : uint8_t {
    NOP = 0x01, HLT = 0x02, ...
    MOV = 0x08, ADD = 0x20, ... };
```

This tells the VM what kinds of instructions it can understand - just like a real CPU understands things like MOV, ADD, PUSH etc, each instruction is given a unique byte number (e.g. MOV=0x08). When your VM reads the program from memory, it interprets each byte based on this list.

Instruction Struct

```
{ opcode op;
  uint16_t a1=0;
  uint16_t a2=0; };
```

eg:- (MOV AX, 5) → (turns to) Instruction {opcode::MOV, 5, 0}

This is defining the basic language that your computer understand

VM class

public: CPU cpu;
Memory memory;
uint16_t breakline = 0; }

This is our full virtual machine - the CPU + mem + program execution logic

→ CPU: has the registers and flags
→ memory: Holds the code and data
→ breakline: used to know where to load the next instruction in memory.

⇒ core methods }
execute: Starts running the instructions (RUN button of PC)
loadProgram: loads a list of instructions into memory.

⇒ Internal logic }
void executeInstruction (const Instruction & instr);
Instruction fetchNextInstruction ();
These help the VM fetch and run instructions one by one, just like how a real CPU reads machine code.

Summary

Component	what it represents	Real world analogy
Registers	CPU's fast internal memory	pockets inside a CPU
CPU	the processor itself	the brain that runs the logic
Memory	RAM/ main memory	like your laptop's ram
opcode	Instruction set	like a dictionary of valid commands
Instruction	A single line of machine code	A commando to execute
VM	The full computer	A working mini-computer in software

2) RohitUtils.hpp

This file contains helper tools (also called "utility functions") that our VM can use to:
→ copy memory
→ clear memory
→ print memory in hex
→ convert between byte orders (networking)
→ Format IP addresses (optional networking support)
These are general-purpose functions, and putting them in a namespace (like a toolkit) keeps your code clean and organized)

pragma once ← makes sure this file is only included once per compilation - avoids duplicate definition errors.

Type aliases
using int8 = uint8_t; → 8 bit number from (0 to 255)
These types are useful for portable code- they always mean exactly the same size on every system

⇒ namespace RohitUtils { ... } :-
This is a container for all your helper functions. By wrapping them in a namespace:-
- you can prevent nameconflicts (like say a function copy() already exists in another files/or lib)
- we can call functions with RohitUtils::copy(...) for clarity

* Function : copy

```
void copy(int8 *dst, const int8* src, int16 size);
```

+ purpose :- copies size bytes from src to dst - similar to memcpy()

+ when to use ?
  → copying values between two memory locations
  → especially used in stack operations (like push or pop in your VM)

+ Example :- Imagine taking 10 Books (bytes) from shelf A and putting them in exact same order in shelf B

Function : nstoh

```
int16 nstoh(int16 srcport);
```

+ purpose : It converts a 16 bit value from network byte order (big-endian) to host byte order (depends on CPU; usually its little endian)

+ why needed :
  - some computers store numbers backword (little endian) while networks use big-endian.
  - This function ensures both ends are speaking the same binary language.

+ Example :- you are reading a number backword (21 as "12"). You fix it so it reads correctly.

Function: zero

```
void zero (int8* str, int16 size);
```

+ purpose: sets a memory block to all zeros.

+ when to use:
  - Before using a memory area (like RAM or a buffer)
  - To "reset" or "clear out" old values.

+ Imagine: you are wiping a whiteboard clean before writing new stuff

Function: printhex

```
void printhex (const int8* str, int16 size, int8 delim = 0);
```

+ purpose: prints raw memory in hexadecimal format like 0A 3F B2

+ why useful:
  - Debugging: Helps you see what's really in memory.
  - understanding what data is stored where

+ Extra: you can choose a delimiter between values (like a space ' ' or a dash '-', etc)

+ Example :- Imagine you are opening ram under a magnifying glass and printing out its contents like detective notes.

* (optional) Function : todotted

const char* todotted (in_addr_t ip);

+ purpose: converts a 32-bit ip address like 0xC0A80101 into human readable form like 192.168.1.1

+ why useful?
  → If we extend our VM for network simulation, this helps display addresses
  → Makes logs and debugs output more human-friendly.

* Summary

Function	what it does	Analogy / use case
copy()	copies bytes from one place to another	xerox machine copying pages
nstoh()	fixes byte order of 16-bit numb.	fixing backward numbers
zero()	sets the memory region to 0	wiping a board clean
printhex()	shows what's inside memory in hex	like viewing RAM under x-ray goggles
todotted()	converts raw IP to "192.168.0.1" format	turning secret code to human readable

☆ This file is short but very important - these helper functions make your VM more readable, debuggable and extendable.

3) RohitUtils.cpp

has implementation of .hpp files

4) RohitVM.cpp

has implementation of .hpp files

ex:- executeInstruction(const Instruction & instr)

```
case opcode::HLT:
    std::cout << "system Halted\n";
    RohitUtils::printHex(...);
    break;

case opcode::ADD: cpu.r.ax += cpu.r.bx; break;
```