MIPS interpreter

Your task is to implement an assembler and interpreter for the MIPS assembly language (32 bit). You can consult https://www.d.umn.edu/~gshute/mips/MIPS.html for a full description of all available instructions, directives and their semantics. We will use a slightly different instruction table based on https://student.cs.uwaterloo.ca/~isg/res/mips/opcodes that is provided as a separate CSV file.

In the base version, you should implement a library core that provides IO functionality for text and binary MIPS assembly, machine state (registers and memory) and instruction execution, together with four binaries `mips-assemble`, `mips-interpret` and `mips-execute` that provide text-to-binary conversion and assembly execution in text or binary form.

As MIPS uses a von Neumann architecture, your machine state consists of the normal registers and a flat memory space that contains both code and data. Since we are using 32 bit memory addresses, you need at most 4 GB of memory to represent the machine state, but we will assume that the memory size is static instead of supporting a `sbrk` syscall.

The `print_*` or `read_*` syscalls should print to `cout` or read data from `cin`, but for testing purposes it may make sense to make the input/output stream configurable, e.g. swapping them out for a `std::stringstream` may help test their behavior.

Description of binary format

Instead of using a fully-featured binary file format like ELF, we'll build our own simplified format. The binary format stores the initial state of memory before executing the program, with the code starting at address `0x0` and the data segment following immediately after. That means that you have to resolve the label names from your text assembly to actual memory addresses, with the `main` label being placed first. You don't need to implement the `.globl` directive, you can assume that all labels are global.

Program behavior

- `mips-assemble` should behave as follows:
  - `mips-assemble` without parameters reads lines of assembly directives and instructions from the standard input (`std::cin`) and writes their binary representation to the standard output (`std::cout`)
  - `mips-assemble input.asm` with a single input filename `input.asm` reads from the given file, and writes the assembled binary representation to the standard output
  - `mips-assemble input.asm output.bin` reads from `input.asm` and writes to `output.bin`
- `mips-interpret` takes a single filename as a parameter, and reads it as a text assembly file. It requires a `main` label and starts executing the assembly code at that position.
- `mips-execute` takes a single filename as a parameter, and reads it as a binary file. It starts executing at the instruction matching the `main` label in the binary file.

How to approach this project

Before starting to write your application, spend some time thinking about a good design for representing all of the aforementioned components of the library core. You should start off by implementing the machine state and some simple instructions from different groups (arithmetic, memory, branching, jumps). In parallel, you can implement the text and binary input and output of individual instructions. If you have to implement a lot of similar instructions, maybe there are some C++ features that can help you?

After you support individual instructions, you should start looking at executing multiple instructions. That means you need to be able to load instructions from memory in their binary form using the program counter register. In parallel, you should implement support for parsing labels and assembly directives. Each instruction or assembly directive reserves a specific amount of memory (e.g. an instruction uses 4 bytes, a `.byte 123` directive uses 1 byte etc.), which can then be used to determine where in memory each value will be stored. Parsing the text assembly is probably the algorithmically most challenging task, so start small by parsing directives and instructions (you only need to implement one parser for each instruction type), and then putting it together to parse an entire assembly source file.

Extensions

Based on the core functionality, you should choose one of the following extensions to your code. You can of course also do multiple extensions, especially some small unit tests will help you avoid bugs more easily.

Extension option 1: Debugger

Implement a debugger based on the library core that reads a text assembly file and allows stepping through the program. It should be able to single-step through your program, and output how the state of the machine changed after each step. More precisely:

- When starting, it shows the first assembly instruction to be executed, and writes a prompt > afterward. It then reads one of three possible commands:
  - `step` executes the currently displayed instruction and moves to and prints the next instruction
  - `reg $name` prints out the value of register `$name`
  - `mem8/mem16/mem32 address` prints out the 8 bit/16 bit/32 bit value in memory at the address.
- You can add additional functionality if you want, e.g. executing until breakpoints at labels or individual instructions

Extension option 2: Unit tests

Use a test framework like Catch or GoogleTest to test as much of your library core as possible. You should test binary and text parsing and output, both for valid and invalid inputs (to test your error handling), and check that the execution of instructions has the intended effect. You don't need to cover all instructions, but you should cover at least one from every group of instructions (arithmetic, memory, branching and jumps).

Extension option 3: Performance optimization

Try to optimize the `mips-execute` binary as much as possible. We will provide a few assembly files that you can use as a benchmark, try to optimize the number of instructions you can execute per second as much as possible.

Assembly instructions and directives

You will be provided a separate list of assembly instructions and directives that should make it easier for you to write or generate the code for them.

Report

In your report, you should describe your software design, especially around the library core, justify design choices and describe what C++ features you utilized. Aim for around one page / 250 words.

Presentation

Your presentation should consist of a short demonstration of your project, focused on either the interpreter/execution of assembly or on the debugger, if you implemented it. Finally, you should give a very brief, 1-2 slide overview over your software design and implementation. In total, aim for a 10 min presentation.

Grading

This project will be graded on functionality, design and code quality. Your priorities should be

1. Your code compiles (ideally without warnings from `-Wall -Wextra`)
2. Your code works correctly for valid MIPS programs
3. Your code is well designed and uses appropriate features of C++
4. Your code does not crash/segfault for invalid inputs, and reports them correctly
5. Your code does not cause any additional failures if compiled with sanitizers enabled
6. Your code is not unnecessarily inefficient, e.g. by making unnecessary copies etc.