Lab 4: Bytecode Virtual Machine

Overview

This project implements a complete toolchain for a custom stack-based bytecode virtual machine (VM). It includes a two-pass assembler built with Flex and Bison to convert a custom assembly language into bytecode, and a VM that loads and executes this bytecode.

Requirements

- G++ (C++17 or newer)
- Flex
- Bison
- Make

Build

To build the entire project, including the VM and the assembler, run:

```
make all
```

This will create the `bvm` (virtual machine) and `assembler` executables in the `build/` directory.

To clean all build files:

```
make clean
```

Usage

Assembler

To assemble an assembly file (`.asm`) into a bytecode file (`.bin`):

```
build/assembler <input_file.asm> <output_file.bin>
```

Virtual Machine

To execute a bytecode file:

```
build/bvm <bytecode_file.bin>
```

You can also run in verbose mode to see a trace of the execution:

```
build/bvm <bytecode_file.bin> --verbose
```

Testing

The project includes three distinct test suites.

1. VM and Component Unit Tests

These are C++ tests for the core components of the VM.

```
make test
```

This target runs tests for the stack, memory, op-codes, and a simple hardcoded VM execution test.

2. Assembler Test Suite

This suite tests the assembler against a set of `.asm` files and compares the output to expected bytecode.

```
make -C Assembler test
```

3. End-to-End Pipeline Tests

This is a comprehensive test suite that assembles and runs a suite of assembly programs, verifying the final stack state for correctness. This is the best way to test the full functionality of the toolchain.

```
make pipeline_test
```

4. Garbage Collector Tests (Lab 5)

This suite verifies the Mark-Sweep GC implementation, including reachability, cycle handling, and stress testing.

```
make test_gc
```

Benchmarks

A set of benchmark programs are included to measure the performance of the VM on different tasks.

```
make benchmark
```

This will run benchmarks for iterative factorial calculation and a simple high-iteration loop, reporting the execution time.

Project Structure

- `src/`: Source code for the VM core (`vm.cpp`, `stack.cpp`, `memory.cpp`, `op_codes.cpp`).
- `test/`: C++ unit tests for the VM components.
- `Assembler/`: Source code for the assembler (`lexer.l`, `parser.y`) and its own test suite.
- `pipeline_tests/`: End-to-end assembly program tests.
- `benchmarks/`: Assembly programs used for performance measurement.
- `Makefile`: Root makefile for building and running all targets.

Features

- **Stack-Based VM**: A virtual machine that uses a stack for all operations.
- **Two-Pass Assembler**: An assembler built with Flex and Bison that supports labels by performing two passes to resolve addresses.
- **Rich Instruction Set**: Includes instructions for data manipulation, arithmetic, bitwise operations, control flow (jumps), memory access, and function calls.
- **Comprehensive Testing**: Includes unit tests, a dedicated assembler test suite, and end-to-end pipeline tests.
- **Benchmarking**: Includes scripts to measure VM performance.

GC Usage

The GC is integrated into the C++ `VM` class.

- **Trigger:** Call `vm.gc()` or the global wrapper `gc(vm)`.
- **Allocation:** Use `vm.new_pair()`, `vm.new_closure()`, etc.
- **Roots:** Objects pushed to the stack using `vm.register_stack.push((long)obj, true)` are treated as roots.