Stack-Based ISA Implementation Documentation

Overview

This project implements a stack-based Instruction Set Architecture (ISA) using C++. The architecture features a virtual machine that executes instructions from an input file, manipulating a data stack and a return stack. The project includes several components: the ALU (Arithmetic Logic Unit), IO/Control (Input/Output) module, and the Machine class that orchestrates the execution and the SDL-based visualization.

1. ALU (Arithmetic Logic Unit)

The ALU performs arithmetic and logical operations on the data stack. It also handles variable storage and retrieval.

Key Implemented Instructions :

1. **push(int operand)**: Pushes an operand onto the data stack.
2. **pushr(int operand):** Pushes an operand onto the return stack.
3. **pop():** Pops the top value from the data stack.
4. **dup():** dulpicates the top value of stack in the data stack.
5. **swap()**: swaps the top two values of the data stack.
6. **add()**: Pops the top two values, adds them, and pushes the result.
7. **inc()**: increases the top value by 1.
8. **dec()**: decreases the top value by 1.
9. **sub()**: Pops the top two values, subtracts the second from the first, and pushes the result.
10. **mul()**: Pops the top two values, multiplies them, and pushes the result.
11. **div()**: Pops the top two values, divides the first by the second, and pushes the result.
12. **mod()**: Pops the top two values, computes the modulus, and pushes the result.
13. **bit_and()**: Pops the top two values, computes the bitwise AND, and pushes the result.
14. **bit_or()**: Pops the top two values, computes the bitwise OR, and pushes the result.
15. **bit_xor()**: Pops the top two values, computes the bitwise XOR, and pushes the result.
16. **bit_not()**: Pops the top value, computes the bitwise NOT, and pushes the result.
17. **cmp()**: Compares the top two values and sets flags for greater than and equal.
18. **gt_jump(int operand)**: Jumps to the specified operand if the greater-than flag is set.

19. **eq_jump(int operand)**: Jumps to the specified operand if the equal flag is set.
20. **jump(int operand)**: Jumps to the specified operand.
21. **ret()**: Returns from a subroutine.
22. **call(int operand)**: Calls a subroutine at the specified operand.
23. **print()**: Prints the top value of the data stack.
24. **halt()**: Halts the execution.
25. **var(int var)**: Declares a variable.
26. **store_var(int var)**: Stores a value in a variable.
27. **load_var(int var)**: Loads a value from a variable.

Working of ALU :

The **alu.hpp** file defines the ALU for the stack-based virtual machine, providing functions to perform arithmetic and logical operations on the data stack. Each function checks the stack for the required number of values, performs the operation, and pushes the result back onto the stack. The ALU operates on the same memory space as the rest of the virtual machine, ensuring consistent and coordinated execution which is ensured by the function **setStacks** which sets the dataStack, returnStack, Program Counter and the memory itself.

2. IO/Control (Input/Output) Module

The IO module handles reading instructions from an input file and initializing the memory.

Key Structures

1. `Instruction`: Represents an instruction with a name and an operand.
2. `var_instruction`: Represents a variable instruction with a name, variable name, and value.

IO Class

```
vector<Instruction> input;
vector<int> output;
map<string, int> label_begins;
map<int, string> labels;
vector<Instruction> *memory;
vector<pair<string, int>> variables;
vector<int> *dataStack;
```

1. **input:** Stores instructions read from the file.
2. **output:** Stores output values.
3. **label_begins:** Maps label names to their positions in the instruction list.
4. **labels:** Maps positions in the instruction list to label names.
5. **memory:** Pointer to a vector of instructions representing memory.
6. **variables:** Stores variable names and their hashed values.

7. **DataStack：** Pointer to a vector representing the data stack.

Key Functions

1. readInput(string filename, vector<Instruction> &mem): Reads instructions from the specified file and populates the memory.

This function reads instructions from a file and processes them:

First Pass:

Reads the file to identify labels and their positions. Strips leading tabs and spaces. Skips empty lines. Identifies labels (lines starting with `.`) and stores their positions in **label_begins** and **labels**.

Second Pass:

Reads the file again to process instructions. Strips leading tabs and spaces. Skips comments (lines starting with `#` or `@`). Processes branch instructions (lines starting with **b**), labels, and other operations.

1. For **push** operations, converts the operand to an integer.
2. For **call** operations, looks up the label's position.
3. For **var**, **store** and **load** operations, hashes the variable name and stores the hash as the operand.

Stores the processed instructions in the **input** vector.

Copy to Memory:

Copies the **input** vector to the provided **memory** vector.

Extra Handling for var, store and load:

For these operations, the code extracts the variable name, hashes it using **hashString**, and stores the hash as the operand. This allows the program to uniquely identify variables by their hashed names.

3. Machine Class

The Machine class initializes the virtual machine, sets up the stacks, and runs the instructions.

Constructor：

Constructor that initializes the machine with the specified memory size and input file.

```
Machine(int memory_size, string filename) {....}
```

1. Initializes the **memory_size**, **memory**, **dataStack**, and **returnStack**.
2. Reads instructions from the file using **ioModule.readInput**.
3. Sets the **stacks** and **program counter** in the ALU using **alu.setStacks**.
4. Sets the **program counter** to the position of the "**main**" label.

Key Functions

run(): Executes the instructions in memory.

Output File: Opens a file to write the output.

Memory Dump: Writes the memory contents to the output file.

Instruction Execution: Executes instructions in a loop until the program counter (`pc`) is -1 or exceeds the instruction list size.

Instruction Handling: Depending on the instruction name, calls the corresponding ALU function.

Data Stack and Return Stack: Writes the contents of the data stack and return stack to the output file.

Program Counter: Updates the program counter after each instruction, except for **halt** and **ret**.

Data Structures

1. Data Stack

The data stack is used for arithmetic and logical operations. Operands are pushed onto the stack, and operations pop operands from the stack, perform the operation, and push the result back onto the stack.

2. Return Stack

The return stack is used for storing return addresses during subroutine calls. When a subroutine is called, the current program counter (PC) is pushed onto the return stack. When the subroutine returns, the address is popped from the return stack and execution continues from that address.

3. Program Counter (PC)

The program counter keeps track of the current instruction being executed. It is incremented after each instruction unless a jump or call instruction modifies it.

Execution Flow

1. **Initialization**:

    i. The **main** function initializes a **Machine** object with the input file.

    ii. The **Machine** constructor reads the input file and sets up the memory and program counter.

2. **Running the Machine**:

    i. The **Run** method of the **Machine** object starts executing instructions.

    ii. For each instruction, it updates the program counter and performs the corresponding operation using the **ALU**.

3. **ALU Operations**:

    i. The **ALU** performs various operations based on the instructions, manipulating the data stack, return stack, and program counter as needed.

4. **Input Handling**:

    i. The **IO/Control** class reads the input file and sets up the instructions and labels.

This setup allows the program to read a set of instructions from a file, execute them using a simulated machine with a stack-based architecture, and output the results for visualization.

Instructions Syntax

Instructions are read from an input file and have the following syntax:

1. **push <value>**: Pushes a value onto the data stack.
2. **pushr<value>:** Pushes an operand onto the return stack.
3. **pop**: Pops the top value from the data stack.
4. **dup:** dulpicates the top value of stack in the data stack.
5. **swap**: swaps the top two values of the data stack.
6. **add**: Adds the top two values on the data stack.
7. **inc**: increases the top value by 1.
8. **dec**: decreases the top value by 1.
9. **sub**: Subtracts the top two values on the data stack.
10. **mul**: Multiplies the top two values on the data stack.
11. **div**: Divides the top two values on the data stack.
12. **mod**: Computes the modulus of the top two values on the data stack.
13. **and**: Computes the bitwise AND of the top two values on the data stack.
14. **or**: Computes the bitwise OR of the top two values on the data stack.

15. **xor**: Computes the bitwise XOR of the top two values on the data stack.
16. **not**: Computes the bitwise NOT of the top value on the data stack.
17. **cmp**: Compares the top two values on the data stack.
18. **bgt <label>**: Jumps to the label if the greater-than flag is set.
19. **beq <label>**: Jumps to the label if the equal flag is set.
20. **b <label>**: Jumps to the label.
21. **call <label>**: Calls a subroutine at the label.
22. **ret**: Returns from a subroutine.
23. **print**: Prints the top value of the data stack.
24. **halt**: Halts the execution.
25. **var <name> <value>**: Declares a variable.
26. **store <name> <value>**: Stores a value in a variable.
27. **load <name>**: Loads a value from a variable.

4. Visualisation Using SDL2

Used **SDL2(**Simple DirectMedia Layer**)** to visualize/simulate the sequential working of the code which makes it working as a **Debugger** too. SDL is a cross platform development library designed to provide low level access to 2D pixel operations, audio, keyboard, mouse, etc.

1. simulates the DataStack in realtime with program counter.
2. simulates returnStack as well containing the return addresses for the subroutine calls.
3. contains button for navigating on the both next and previous instructions.