Project6 for Intro to Computer Systems 2025 spring

Yiming Cheng

12450588

setup

How to Compile the Program

1. **Open the Program Folder**: First, you need to go to the folder where the program files are stored. This folder is named `src/`. If you don't know how to get there, ask someone to help you open the folder.

   ```
   cd path/to/src
   ```

   make sure you can see

   ```
   ls
   ```

```
.
├── BasicTest.vm
├── PointerTest.vm
├── SimpleAdd.vm
├── StackTest.vm
├── StaticTest.vm
└── VMT.java
```

2. **Open the Command Line**:

- On Windows, press `Win + R`, type `cmd`, and press Enter to open the Command Prompt.

- On macOS or Linux, open the Terminal app.

3. **Check if Java is Installed**: In the Command Prompt or Terminal, type the following and press Enter:

```
javac -version
```

If you see something like javac 1.8.x, then Java is installed, and you're ready to proceed.

If it shows an error, Java might not be installed, and you'll need to download it from here.

4.**Compile the Program**

```
javac VMT.java
ls
```

Now you can see

```
.
├── BasicTest.vm
├── PointerTest.vm
├── SimpleAdd.vm
├── StackTest.vm
├── StaticTest.vm
├── VMT$Lex.class
├── VMT$Writer.class
├── VMT.class
└── VMT.java
```

5 run the translator

```
java VMT StaticTest.vm
java VMT PointerTest.vm
java VMT SimpleAdd.vm
java VMT StackTest.vm
java VMT BasicTest.vm
ls
```

make sure you can see

```
.
├── BasicTest.asm
├── BasicTest.vm
├── PointerTest.asm
├── PointerTest.vm
```

```
├── SimpleAdd.asm
├── SimpleAdd.vm
├── StackTest.asm
├── StackTest.vm
├── StaticTest.asm
├── StaticTest.vm
├── VMT$Lex.class
├── VMT$Writer.class
├── VMT.class
└── VMT.java
```

This code implements a basic VM-to-Assembly code translator, which converts VM commands into their corresponding Assembly language instructions. Below is a detailed explanation:

1. **Lex**

The `Lex` class is responsible for reading and parsing the VM input file. It breaks down the input commands and categorizes them into different types (e.g., arithmetic operations, push/pop, function calls, etc.). Here's how it works:

- **Attributes**:
  - `cmds`: A `Scanner` object used to read the input file line by line.
  - `currentCmd`: Stores the current command being processed.
  - `argType`, `argument1`, and `argument2`: Represent the type of the current command and its arguments.
  - `arithmeticCmds`: A static list of arithmetic commands (add, sub, etc.).

- **Methods**:
  - `Lex(File fileIn)`: This constructor opens the input file and processes it, removing any comments (lines starting with `//`) and then sets up the file for parsing.
  - `hasMoreCommands()`: Returns true if there are more commands to process.
  - `advance()`: Reads the next command, splits it into segments, and categorizes it.
  - `commandType()`: Returns the type of the current command (arithmetic, push, pop, etc.).
  - `arg1()`: Returns the first argument of the current command (or throws an error for return commands).
  - `arg2()`: Returns the second argument for certain commands like push/pop.
  - `noComments(String strIn)`: Removes comments from the input string.
  - `noSpaces(String strIn)`: Removes spaces from the input string.
  - `getExt(String fileName)`: Gets the file extension from a filename.

2. **Writer**

The `Writer` class is responsible for generating the Assembly code based on the parsed VM commands. It writes the translated Assembly code to an output file.

- **Attributes**:

- `arthJumpFlag`: A flag used to create unique jump labels for conditional arithmetic operations.
- `outPrinter`: A `PrintWriter` used to write to the output file.

- **Methods**:
  - `setFileName(File fileOut)`: Sets the file name for the output file.
  - `writeArithmetic(String command)`: Generates Assembly code for arithmetic operations (add, sub, neg, eq, etc.).
  - `writePushPop(int command, String segment, int index)`: Generates Assembly code for push and pop commands.
  - `close()`: Closes the output file.
  - Helper methods like `arithmeticTemplate1()`, `arithmeticTemplate2()`, `pushTemplate1()`, and `popTemplate1()` are used to generate specific parts of Assembly code for various commands.

3. **Main Program**

The `main` method is the entry point of the program. It processes input and output files and orchestrates the translation from VM to Assembly.

- **Steps**:
  - First, it checks if the input is a file or a directory.
  - If it's a single `.vm` file, it processes that file.
  - If it's a directory, it collects all `.vm` files within it.

    - It then uses the `Lex` class to parse each file and the `Writer` class to generate the Assembly code.
    - After processing all commands, the generated Assembly code is written to the output file.
- **Error Handling**:
    - The program checks if the input is a valid `.vm` file or directory, ensuring that only `.vm` files are processed.
    - If invalid commands or arguments are found, the program throws an exception.

4. **Getting `.vm` Files in a Directory**

The `getVMFiles()` method scans a directory for `.vm` files. If any are found, it returns them as a list.

Example VM-to-Assembly Translation

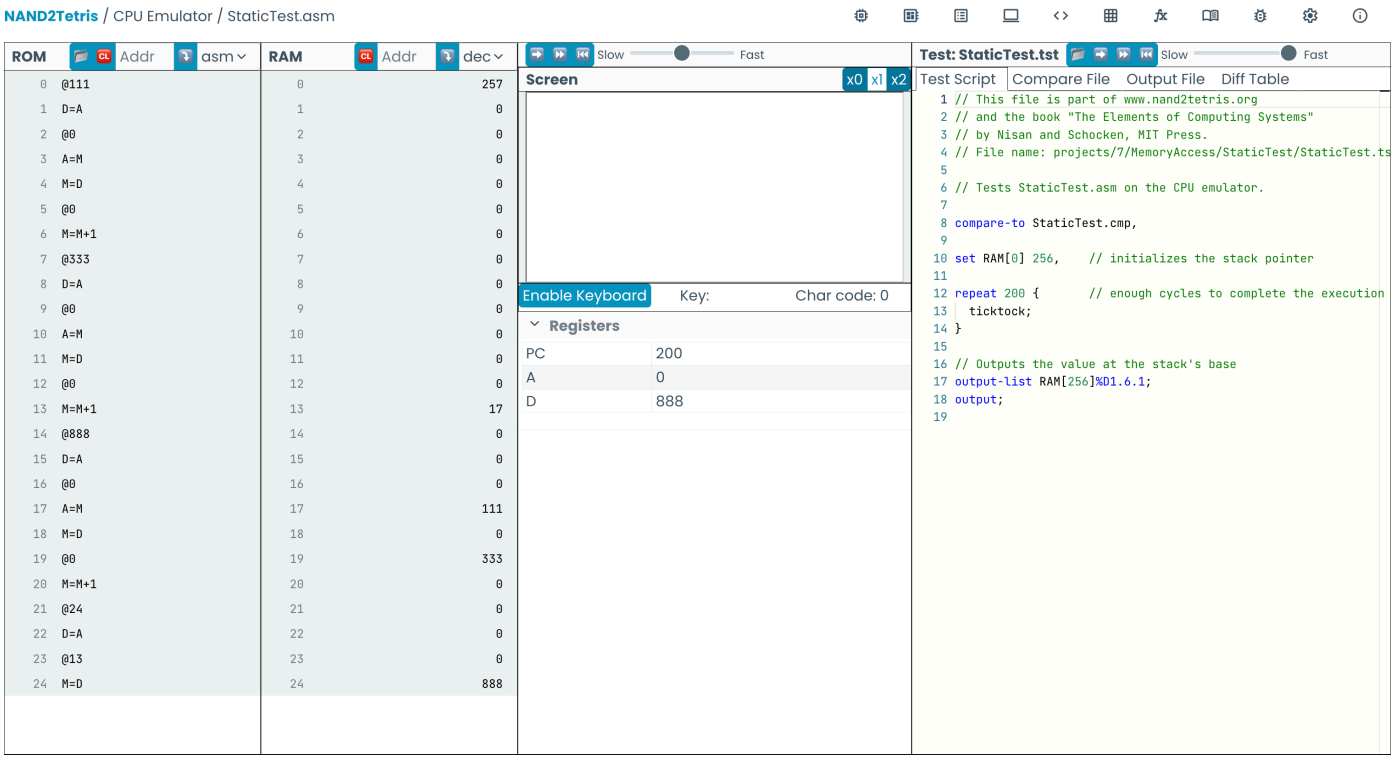
```
push constant 10
```

```
@10
D=A
@SP
A=M
M=D
@SP
M=M+1
```

tests

All five asm code pass 100%

you can try either the compiled .asm files in the src or try it in online IDE



Simulation successful: The output file is identical to the compare file