Contents

1 Basic Test Results

```
********** TESTING FOLDER STRUCTURE START **********
Running test8.sh:
Checking your submission for presence of invalid (non-ASCII) characters...
No invalid characters found.

Your logins are: muaz.abdeen, is that ok?

********** TESTING FOLDER STRUCTURE END **********

********** PROJECT TEST START **********

Running: 'make'
chmod a+x VMtranslator
Running your program with command: './VMtranslator tst/FibonacciElement'
Assembler translated FibonacciElement.asm
Execution of FibonacciElement.asm was successful

********** PROJECT TEST END **********
```

2 README

```
muaz.abdeen
===============================================================================
Muaz Abdeen, ID 300575297, muaz.abdeen@mail.huji.ac.il
===============================================================================

                           Project 8 - VM II: Program Control
                           ----------------------------------


Submitted Files
---------------
(1)  README                 - This file.
(2)  VMParser.py        - The VMParser module implementation.
(3)  VMCodeWriter.py        - The VMCodeWriter module implementation.
(4)  VMtranslator.py        - The VMtranslator module implementation.
(5)  VMtranslator            - The run file.
(6)  Makefile                - The make file.


Remarks
-------
* ...
```

3 Makefile

```
all:
    chmod a+x VMtranslator
```

4 VMCodeWriter.py

```python
################################################
##   FILE : VMCodeWriter.py                   ##
##   WRITER : muaz.abdeen, 300575297          ##
##   EXERCISE : nand2tetris project07 2020A   ##
################################################

import os
import VMParser


class VMCodeWriter:
    """
    Translates VM commands into Hack assembly code.
    """

    ##########################
    ##  MACROS & CONSTANTS  ##
    ##########################

    ###################
    ##  CONSTRUCTOR  ##
    ###################

    def __init__(self, output_file):
        """
        Opens the output file and gets ready to write it.
        :param output_file: name of the file to write to (.asm) file
        :type output_file: str
        """
        self.output_file = open(output_file, mode='w')
        self.current_VMfile = ''
        self.comparison_counter = 0
        self.current_function = 'null'
        self.call_counter = 0
        self.writeInit()
        # End of Constructor

    ###############
    ##  METHODS  ##
    ###############

    def setFileName(self, file_name):
        """
        Informs the code writer that the translation of a new VM file is started.
        (called by the main program of the VM translator)
        :param file_name: the new file to be translated
        :type file_name: str
        :return: None
        """
        self.current_VMfile = os.path.basename(file_name)

    def writeArithmetic(self, command):
        """
        Writes the assembly code that is the translation of the given arithmetic command.
        :param command: an arithmetic command.
        :type command: str
        :return: None
        """
        asm_commands = ''
        # binary arithmetic
        if command == 'add':
            asm_commands = VMCodeWriter._binaryArithmetic('+')
        elif command == 'sub':
            asm_commands = VMCodeWriter._binaryArithmetic('-')
        elif command == 'and':
            asm_commands = VMCodeWriter._binaryArithmetic('&')
        elif command == 'or':
            asm_commands = VMCodeWriter._binaryArithmetic('|')

        # unary arithmetic and logical
        elif command == 'neg':
            asm_commands = VMCodeWriter._unaryArithmeticOrLogical('-')
        elif command == 'not':
            asm_commands = VMCodeWriter._unaryArithmeticOrLogical('!')

        # binary logical
        elif command == 'eq':
            asm_commands = self._binaryLogical('JNE')
            # asm_commands = VMCodeWriter._binaryLogical('JEQ')
        elif command == 'gt':
            asm_commands = self._binaryLogical('JGE')
        elif command == 'lt':
            asm_commands = self._binaryLogical('JLE')

        self.output_file.write(f'// {command}\n' + asm_commands)
        # End of writeArithmetic() method

    @staticmethod
    def _binaryArithmetic(operator):
        asm_commands = f'  @SP\n' \
                       f'  AM=M-1\n' \
                       f'  D=M\n' \
                       f'  A=A-1\n'
        if operator == '-':
            return asm_commands + f'  M=M-D\n'
        # the order of commutative operations as appears in the book (D <operator> M)
        return asm_commands + f'  M=D{operator}M\n'

    @staticmethod
    def _unaryArithmeticOrLogical(operator):
        return f'  @SP\n' \
               f'  A=M-1\n' \
               f'  M={operator}M\n'

    def _binaryLogical(self, jump):
        res = (-1, 0) if jump == 'JGE' else (0, -1)
        asm_command = f'  @SP\n' \
                      f'  AM=M-1\n' \
                      f'  D=M\n' \
                      f'  // check if y<0\n' \
                      f'  @Y_NG_{self.comparison_counter}\n' \
                      f'  D;JLT     // y<0 \n' \
                      f'  // check if x<0 \n' \
                      f'  @SP\n' \
                      f'  A=M-1\n' \
                      f'  D=M\n' \
                      f'  @X_NG_{self.comparison_counter}\n' \
                      f'  D;JLT     // x<0 \n' \
                      f'(SAME_SIGN_{self.comparison_counter})\n' \
                      f'  @SP\n' \
                      f'  A=M\n' \
                      f'  D=M\n' \
                      f'  A=A-1\n' \
                      f'  D=D-M\n' \
                      f'  M=-1\n' \
                      f'  @FALSE_{self.comparison_counter}\n' \
                      f'  D;{jump}\n' \
                      f'  @END_{self.comparison_counter}\n' \
                      f'  0;JMP\n' \
                      f'(FALSE_{self.comparison_counter})\n' \
                      f'  @SP\n' \
                      f'  A=M-1\n' \
                      f'  M=0\n' \
                      f'  @END_{self.comparison_counter}\n' \
                      f'  0;JMP\n' \
                      f'(Y_NG_{self.comparison_counter})\n' \
                      f'  // check if x >= 0 \n' \
                      f'  @SP\n' \
                      f'  A=M-1\n' \
                      f'  D=M\n' \
                      f'  @SAME_SIGN_{self.comparison_counter}\n' \
                      f'  D;JLT\n' \
                      f'  @SP\n' \
                      f'  A=M-1\n' \
                      f'  M={res[0]}      // y<0 , x>=0 \n' \
                      f'  @END_{self.comparison_counter}\n' \
                      f'  0;JMP\n' \
                      f'(X_NG_{self.comparison_counter})\n' \
                      f'  @SP\n' \
                      f'  A=M-1\n' \
                      f'  M={res[1]}      // y>=0 , x<0 \n' \
                      f'(END_{self.comparison_counter})\n'
        self.comparison_counter += 1
        return asm_command

    def writePushPop(self, command, segment, index):
        """
        Writes the assembly code that is the translation of the given command,
        where command is either C_PUSH or C_POP.
        :param command: a C_PUSH or C_POP command.
        :type command: int
        :param segment: the memory segment write to or from.
        :type segment: str
        :param index: the index of the memory word
        :type index: int
        :return: None
        """
        segments_map = {'local': 'LCL', 'argument': 'ARG', 'this': 'THIS', 'that': 'THAT'}
        command_map = {VMParser.VMParser.C_PUSH: 'push', VMParser.VMParser.C_POP: 'pop'}
        pre_comment = f'// {command_map[command]} {segment} {index}\n'
        asm_commands = ''
        if command == VMParser.VMParser.C_PUSH:
            if segment == 'constant':
                asm_commands = VMCodeWriter._pushConstant(index)
            elif segment in segments_map:
                asm_commands = VMCodeWriter._pushSegment1(segments_map[segment], index)
            elif segment in {'temp', 'pointer', 'static'}:
                asm_commands = self._pushSegment2(segment, index)

        elif command == VMParser.VMParser.C_POP:
            if segment in segments_map:
                asm_commands = VMCodeWriter._popSegment1(segments_map[segment], index)
            elif segment in {'temp', 'pointer', 'static'}:
                asm_commands = self._popSegment2(segment, index)

        self.output_file.write(pre_comment + asm_commands)
        # End of writePushPop() method

    @staticmethod
    def _pushConstant(index):
        return f'  @{index}\n' \
               f'  D=A\n' \
               f'  @SP\n' \
               f'  AM=M+1\n' \
               f'  A=A-1\n' \
```

```python
                 f'  M=D\n'

    @staticmethod
    def _popSegment1(segment, index):
        return f'  @{segment}\n' \
               f'  D=M\n' \
               f'  @{index}\n' \
               f'  D=D+A\n' \
               f'  @R13\n' \
               f'  M=D\n' \
               f'  @SP\n' \
               f'  AM=M-1\n' \
               f'  D=M\n' \
               f'  @R13\n' \
               f'  A=M\n' \
               f'  M=D\n'

    @staticmethod
    def _pushSegment1(segment, index):
        return f'  @{segment}\n' \
               f'  D=M\n' \
               f'  @{index}\n' \
               f'  A=A+D\n' \
               f'  D=M\n' \
               f'  @SP\n' \
               f'  AM=M+1\n' \
               f'  A=A-1\n' \
               f'  M=D\n'

    def _popSegment2(self, segment, index):
        if segment == 'pointer':
            label = 'THIS' if not index else 'THAT'
        elif segment == 'temp':
            label = index + 5
        else:  # static
            label = f'{os.path.split(self.current_VMfile)[1][:-3]}.{index}'
        return f'  @SP\n' \
               f'  AM=M-1\n' \
               f'  D=M\n' \
               f'  @{label}\n' \
               f'  M=D\n'

    def _pushSegment2(self, segment, index):
        if segment == 'pointer':
            label = 'THIS' if not index else 'THAT'
        elif segment == 'temp':
            label = index + 5
        else:  # static
            label = f'{os.path.split(self.current_VMfile)[1][:-3]}.{index}'
        return f'  @{label}\n' \
               f'  D=M\n' \
               f'  @SP\n' \
               f'  AM=M+1\n' \
               f'  A=A-1\n' \
               f'  M=D\n'

    def writeInit(self):
        """
        Writes assembly code that effects the VM initialization, also called bootstrap code.
        This code must be placed at the beginning of the output file.
        :return: None
        """
        pre_comment = '// Initializing the VM program\n'
        asm_command = f'  @256\n' \
                      f'  D=A\n' \
                      f'  @SP\n' \
                      f'  M=D\n'
        self.output_file.write(pre_comment + asm_command)
        self.writeCall('Sys.init', 0)

    def writeLabel(self, label):
        """
        Writes assembly code that effects the label command.
        :param label: a given label
        :type label: str
        :return: None
        """
        self.output_file.write(f'({self.current_function}${label})\n')

    def writeGoto(self, label):
        """
        Writes assembly code that effects the goto command.
        :param label: a label with goto command
        :type label: str
        :return: None
        """
        self.output_file.write(f'  @{self.current_function}${label}\n'
                               f'  0;JMP\n')

    def writeIf(self, label):
        """
        Writes assembly code that effects the if-goto command.
        :param label: a label with if-goto command
        :type label: str
        :return: None
        """
        self.output_file.write(f'// if-goto\n'
                               f'  @SP\n'
                               f'  M=M-1\n'
                               f'  A=M\n'
                               f'  D=M\n'
                               f'  @{self.current_function}${label}\n'
                               f'  D;JNE\n')

    def writeCall(self, function_name, num_args):
        """
        Writes assembly code that effects the call command.
        :param function_name: the name of the callee
        :type function_name: str
        :param num_args: number of the callee arguments
        :type num_args: int
        :return: None
        """
        pre_comment = f'  // call {function_name} {num_args}\n'
        asm_command = f'  // push retAddrLabel \n' + \
                      VMCodeWriter._pushConstant(f'{function_name}$RETURN_{self.call_counter}')
        for label in ['LCL', 'ARG', 'THIS', 'THAT']:
            asm_command += f'  // push {label}\n' \
                           f'  @{label}\n' \
                           f'  D=M\n' \
                           f'  @SP\n' \
                           f'  M=M+1\n' \
                           f'  A=M-1\n' \
                           f'  M=D\n'
        asm_command += f'  // ARG = SP-5-nArgs \n' \
                       f'  @SP\n' \
                       f'  D=M\n' \
                       f'  @5\n' \
                       f'  D=D-A\n' \
                       f'  @{num_args}\n' \
                       f'  D=D-A\n' \
                       f'  @ARG\n' \
                       f'  M=D\n' \
                       f'  // LCL = SP\n' \
                       f'  @SP\n' \
                       f'  D=M\n' \
                        f'  @LCL\n' \
                        f'  M=D\n' \
                        f'  // goto functionName\n' \
                        f'  @{function_name}\n' \
                        f'  0;JMP\n' \
                        f'({function_name}$RETURN_{self.call_counter})\n'
        self.call_counter += 1
        self.output_file.write(pre_comment + asm_command)
        # End of writeCall() method

    def writeReturn(self):
        """
        Writes assembly code that effects the return command.
        :return: None
        """
        pre_comment = '  // return\n'
        asm_command = f'  // endFrame (R13) = LCL\n' \
                      f'  @LCL\n' \
                      f'  D=M\n' \
                      f'  @R13\n' \
                      f'  MD=D\n' \
                      f'  // retAddr (R14) = *(endFrame - 5) \n' \
                      f'  @5\n' \
                      f'  A=D-A\n' \
                      f'  D=M\n' \
                      f'  @R14\n' \
                      f'  M=D\n' \
                      f'  // *ARG=pop() \n' \
                      f'  @SP\n' \
                      f'  A=M-1\n' \
                      f'  D=M\n' \
                      f'  @ARG\n' \
                      f'  A=M\n' \
                      f'  M=D\n' \
                      f'  // SP = ARG + 1\n' \
                      f'  D=A\n' \
                      f'  @SP\n' \
                      f'  M=D+1\n'
        idx = 1
        for label in ['THAT', 'THIS', 'ARG', 'LCL']:
            asm_command += f'  // {label} = *(endFrame-{idx}) \n' \
                           f'  @R13\n' \
                           f'  D=M\n' \
                           f'  @{idx}\n' \
                           f'  A=D-A\n' \
                           f'  D=M\n' \
                           f'  @{label}\n' \
                           f'  M=D\n'
            idx += 1
        asm_command += f'  // goto retAddr \n' \
                       f'  @R14\n' \
                       f'  A=M\n' \
                       f'  0;JMP\n'

        self.output_file.write(pre_comment + asm_command)
        # End of writeReturn() method

    def writeFunction(self, function_name, num_locals):
        """
        Writes assembly code that effects the function command.
        :param function_name: the name og the function
        :type function_name: str
        :param num_locals: number of local variables of the function
        :type num_locals: int
        :return: None
        """
        self.current_function = function_name
        pre_comment = f' // function {function_name} {num_locals}\n'
        asm_command = f'({function_name})\n'
        if num_locals:
            asm_command += f'  @LCL\n' \
                           f'  A=M\n'
            for n in range(num_locals):
                asm_command += f'  M=0\n' \
                               f'  A=A+1\n'
            asm_command += f'  D=A\n' \
                           f'  @SP\n' \
                           f'  M=D\n'
        self.output_file.write(pre_comment + asm_command)
        # End of writeFunction() method

    def close(self):
        """
        Closes the output file.
        :return: None
        """
        self.output_file.close()

    # End of VMCodeWriter class
```

5 VMParser.py

```python
########################################################
##   FILE : VMParser.py                               ##
##   WRITER : muaz.abdeen, 300575297                  ##
##   EXERCISE : nand2tetris projects07-08 2020A       ##
########################################################


class VMParser:
    """
    Handles the parsing of a single .vm file, and encapsulates access to the input code.
    It reads VM commands, parses them, and provides convenient access to their components.
    In addition, it removes all white space and comments.
    """

    ##########################
    ##  MACROS & CONSTANTS  ##
    ##########################

    NOT_FOUND = -1

    C_ARITHMETIC = 1
    C_PUSH = 2
    C_POP = 3
    C_LABEL = 4
    C_GOTO = 5
    C_IF = 6
    C_FUNCTION = 7
    C_CALL = 8
    C_RETURN = 9

    _arithmetic_commands = ['add', 'sub', 'neg', 'eq', 'gt', 'lt', 'and', 'or', 'not']
    _type_map = {'push': C_PUSH, 'pop': C_POP,
                 'label': C_LABEL, 'goto': C_GOTO, 'if-goto': C_IF,
                 'function': C_FUNCTION, 'call': C_CALL, 'return': C_RETURN}

    ###################
    ##  CONSTRUCTOR  ##
    ###################

    def __init__(self, file_name):
        """
        Opens the input file and gets ready to parse it.
        :param file_name: name of the file to parse (.vm) file
        :type file_name: str
        """
        self.file = open(file_name, mode='r')
        self.command = ''
        # End of Constructor

    ###############
    ##  METHODS  ##
    ###############

    def hasMoreCommands(self):
        """
        Are there more commands in the input?
        :return: True if there are more commands, False else
        :rtype: bool
        """
        return self.file is not None

    def advance(self):
        """
        Reads the next command from the input and makes it the current command.
        Should be called only if hasMoreCommands() is true.
        Initially there is no current command.
        :return: None
        """
        self.command = self.file.readline()
        if not self.command:
            self.file.close()
            self.file = None
            return
        # deals with comments
        comment_idx = self.command.find('//')
        if comment_idx != self.NOT_FOUND:  # the line contains a comment
            self.command = self.command[:comment_idx]
        # list of command parts
        self.command = self.command.split()
        # blank or pure comment line
        if not self.command:
            self.advance()
        # End of advance() method

    def commandType(self):
        """
        Returns the type of the current VM command.
        C_ARITHMETIC is returned for all the arithmetic commands.
        :return: the current command type
        :rtype: int
        """
        if self.command[0] in VMParser._arithmetic_commands:
            return self.C_ARITHMETIC
        return VMParser._type_map[self.command[0]]
        # End of commandType() method

    def arg1(self):
        """
        Returns the first argument of the current command.
        In the case of C_ARITHMETIC, the command itself ("add", "sub", etc.) is returned.
        Should not be called for C_RETURN.
        :return: the first argument of the current command.
        :rtype: str
        """
        assert self.commandType() != VMParser.C_RETURN
        if self.commandType() == VMParser.C_ARITHMETIC:
            return self.command[0]
        return self.command[1]

    def arg2(self):
        """
        Returns the second argument of the current command.
        Should be called only if the current command is C_PUSH, C_POP, C_FUNCTION, or C_CALL.
        :return: the second argument of the current command.
        :rtype: int
        """
        assert len(self.command) == 3
        return int(self.command[2])

    # End of VWParser class
```

6 VMtranslator

```
#!/bin/sh
python3 VMtranslator.py $*
```

7 VMtranslator.py

```
################################################
##   FILE : VMtranslator.py                   ##
##   WRITER : muaz.abdeen, 300575297          ##
##   EXERCISE : nand2tetris project07 2020A   ##
################################################

import VMParser
import VMCodeWriter
import sys
import os


def VMtranslator(input_file, code_writer):
    """
    translates the VM code file into assembly language file
    :param input_file: VM code file
    :type input_file: str
    :param code_writer: the code writer
    :type code_writer: VMCodeWriter.VMCodeWriter
    :return: None
    """
    parser = VMParser.VMParser(input_file)
    while parser.hasMoreCommands():
        parser.advance()
        if parser.command:
            command_type = parser.commandType()
            if command_type in {parser.C_POP, parser.C_PUSH}:
                code_writer.writePushPop(command_type, parser.arg1(), parser.arg2())
            elif command_type == parser.C_ARITHMETIC:
                code_writer.writeArithmetic(parser.arg1())
            elif command_type == parser.C_LABEL:
                code_writer.writeLabel(parser.arg1())
            elif command_type == parser.C_IF:
                code_writer.writeIf(parser.arg1())
            elif command_type == parser.C_GOTO:
                code_writer.writeGoto(parser.arg1())
            elif command_type == parser.C_FUNCTION:
                code_writer.writeFunction(parser.arg1(), parser.arg2())
            elif command_type == parser.C_CALL:
                code_writer.writeCall(parser.arg1(), parser.arg2())
            elif command_type == parser.C_RETURN:
                code_writer.writeReturn()
    # End of VMtranslator() function


def main():
    if len(sys.argv) != 2:
        print("Usage: VMtranslator <file.vm or path>")
        sys.exit(-1)

    program_input = sys.argv[1]
    if os.path.isdir(program_input):
        output_file = os.path.split(program_input)[1] + os.path.extsep + 'asm'
        output_path = program_input + os.path.sep + output_file
        code_writer = VMCodeWriter.VMCodeWriter(output_path)
        for entry in os.scandir(program_input):
            if entry.is_file() and entry.name.endswith('.vm'):
                code_writer.setFileName(entry.name)
                full_name = os.path.join(os.path.abspath(program_input), entry.name)
                VMtranslator(full_name, code_writer)
        code_writer.close()

    elif program_input.endswith('.vm'):
        code_writer = VMCodeWriter.VMCodeWriter(program_input.replace('vm', 'asm'))
        code_writer.setFileName(program_input)
        VMtranslator(program_input, code_writer)
        code_writer.close()


if __name__ == '__main__':
    main()
```