Contents

1 Basic Test Results 2

2 README 3

3 Assembler 4

4 Code.py 5

5 Main.py 7

6 Makefile 10

7 Parser.py 11

8 SymbolTable.py 14

1 Basic Test Results

```
********** TESTING FOLDER STRUCTURE START **********
Running test6.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 **********

chmod a+x Assembler
Running your program with the following command: ./Assembler Add.asm
The diff succeeded on the file Add.hack
Running your program with the following command: ./Assembler tst/Add.asm
The diff succeeded on the file tst/Add.hack

If one of the tests didn't work, make sure you're handling all the various path
options (single files, folders, etc') correctly. See moodle submission page for
more info.

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

2 README

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

                           Project 6 - Assembler
                           ---------------------


Submitted Files
---------------
(1)  README         - This file.
(2)  Parser.py        - The Parser module implementation.
(3)  Code.py        - The Code module implementation.
(4)  SymbolTable.py    - The Symbol Table implementation.
(5)  Main.py        - The Assembler implementation.
(6)  Assembler        - The run file.
(7)  Makefile        - The MakeFile.


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

3 Assembler

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

4 Code.py

```python
################################################
##   FILE : Code.py                           ##
##   WRITER : muaz.abdeen, 300575297          ##
##   EXERCISE : nand2tetris project06 2020A   ##
################################################

class Code:
    """
    Class that codes the parsed assembly command into its binary representation.
    """

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

    _destinations = ['A', 'D', 'M']

    _jumpMap = {
        '': '000',
        'JGT': '001',
        'JEQ': '010',
        'JGE': '011',
        'JLT': '100',
        'JNE': '101',
        'JLE': '110',
        'JMP': '111'
    }

    _compMap = {
        '0': '0101010',
        '1': '0111111',
        '-1': '0111010',
        'D': '0001100',
        'A': '0110000',   'M': '1110000',
        '!D': '0001101',
        '!A': '0110001',  '!M': '1110001',
        '-D': '0001111',
        '-A': '0110011',  '-M': '1110011',
        'D+1': '0011111',
        'A+1': '0110111', 'M+1': '1110111',
        'D-1': '0001110',
        'A-1': '0110010', 'M-1': '1110010',
        'D+A': '0000010', 'D+M': '1000010',
        'D-A': '0010011', 'D-M': '1010011',
        'A-D': '0000111', 'M-D': '1000111',
        'D&A': '0000000', 'D&M': '1000000',
        'D|A': '0010101', 'D|M': '1010101',
        'D>>': '0010000',
        'D<<': '0110000',
        'A>>': '0000000',
        'A<<': '0100000',
                           'M>>': '1000000',
                           'M<<': '1100000',
    }

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

    @staticmethod
    def dest(mnemonic):
        """
        Returns the binary code of the dest mnemonic.
        :param mnemonic: the dest mnemonic.
        :type mnemonic: str
        :return: the binary code of the dest mnemonic.
        :rtype: str
        """
        dest_code = ''
        for dest in Code._destinations:
            dest_code += '1' if dest in mnemonic else '0'
        return dest_code

    @staticmethod
    def comp(mnemonic):
        """
        Returns the binary code of the comp mnemonic.
        :param mnemonic: the comp mnemonic.
        :type mnemonic: str
        :return: the binary code of the comp mnemonic.
        :rtype: str
        """
        return Code._compMap[mnemonic]

    @staticmethod
    def jump(mnemonic):
        """
        Returns the binary code of the jump mnemonic.
        :param mnemonic: the jump mnemonic.
        :type mnemonic: str
        :return: the binary code of the jump mnemonic.
        :rtype: str
        """
        return Code._jumpMap[mnemonic]

    # End of Code class
```

5 Main.py

```python
##################################################
##   FILE : Main.py                             ##
##   WRITER : muaz.abdeen, 300575297            ##
##   EXERCISE : nand2tetris project06 2020A     ##
##################################################

import Code
import Parser
import SymbolTable
import sys
import os


class Assembler:
    """
    The assembler program that translates an assembly language command to its
    binary representation.
    """

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

    """
    The RAM address that variables are initially mapped to.
    """
    INIT_VARIABLES_ADDRESS = 16

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

    def __init__(self):
        """
        Initializes the symbol table associated with this assembler, and
        the memory address to start mapping variables from it.
        """
        self.symbolTable = SymbolTable.SymbolTable()
        self.variableAddress = Assembler.INIT_VARIABLES_ADDRESS

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

    # FIRST PASS :
    def firstPass(self, input_file):
        """
        Go through the entire assembly program, line by line, and build
        the symbol table without generating any code.
        The program's variables are handled in the second pass.
        :param input_file: the input file name
        :type input_file: str
        :return: None
        """
        parser = Parser.Parser(input_file)
        command_address = 0
        while parser.hasMoreCommands():
            parser.advance()
            if parser.command:
                command_type = parser.commandType()
                if command_type == parser.A_COMMAND or command_type == parser.C_COMMAND:
                    command_address += 1
                elif command_type == parser.L_COMMAND:
                    self.symbolTable.addEntry(parser.symbol(), command_address)

    # SECOND PASS :
    def secondPass(self, input_file, output_file):
        """
        Go again through the entire program, and parse each line.
        Adding new variables to the symbol table.
        :param input_file: input file name
        :type input_file: str
        :param output_file: output file name
        :type output_file: str
        :return: None
        """
        parser = Parser.Parser(input_file)
        with open(output_file, mode='w') as hack_file:
            while parser.hasMoreCommands():
                parser.advance()
                if parser.command:
                    command_type = parser.commandType()
                    if command_type == parser.A_COMMAND:
                        A_instruction = self._getAInstruction(parser.symbol())
                        hack_file.write(A_instruction)
                    elif command_type == parser.C_COMMAND:
                        C_instruction = self._getCInstruction(parser.comp(), parser.dest(),
                                                              parser.jump())
                        hack_file.write(C_instruction)

    def _getAInstruction(self, symbol):
        """
        Get the A_Instruction of the given symbol in binary representation
        :param symbol: the symbol to get its A_Instruction
        :type symbol: str
        :return: the binary representation of the symbol
        :rtype: str
        """
        # instruction = ''
        if symbol.isdigit():
            instruction = symbol
        else:
            if not self.symbolTable.contains(symbol):
                self.symbolTable.addEntry(symbol, self.variableAddress)
                self.variableAddress += 1
            instruction = self.symbolTable.getAddress(symbol)
        return f'{bin(int(instruction))[2:].zfill(16)}\n'

    def _getCInstruction(self, comp, dest, jump):
        """
        Get the C_Instruction of the given sub-instructions.
        :param comp: comp command
        :type comp: str
        :param dest: dest command
        :type dest: str
        :param jump: jump command
        :type jump: str
        :return: the C_Instruction of the given sub-instructions.
        :rtype: str
        """
        op_code = '101' if '<<' in comp or '>>' in comp else '111'
        comp_code = Code.Code.comp(comp)
        dest_code = Code.Code.dest(dest)
        jump_code = Code.Code.jump(jump)
        return f'{op_code}{comp_code}{dest_code}{jump_code}\n'

    def executeAll(self, input_file):
        """
        Carry out all the assemble operations
        :param input_file: An assembly language file
        :type input_file: str
        :return: None
        """
        self.firstPass(input_file)
        self.secondPass(input_file, input_file.replace('.asm', '.hack'))

    # End of Assembler class


def main():
    if len(sys.argv) != 2:
        print("Usage: Assembler <file.asm>")
        sys.exit(-1)

    assembler = Assembler()
    program_input = sys.argv[1]
    if os.path.isdir(program_input):
        for entry in os.scandir(program_input):
            if entry.is_file() and entry.name.endswith('.asm'):
                full_name = os.path.join(os.path.abspath(program_input), entry.name)
                assembler.executeAll(full_name)
    elif program_input.endswith('.asm'):
        assembler.executeAll(program_input)


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

6 Makefile

```
###############################################################################
#
# Makefile for a script (e.g. Python), project 6
#
###############################################################################

# **** Why do we need this file? ****
# We want our users to have a simple API to run the Assembler, no matter the language
# it was written in. So, we need a "wrapper" that will hide all language-specific details to do so,
# thus enabling our users to simply type 'Assembler <path>' in order to use it.

# **** What are makefiles? ****
# This is a sample makefile.
# The purpose of makefiles is to make sure that after running "make" your project is ready for execution.

# **** What should I change in this file to make it work with my project? ****
# Usually, scripting language (e.g. Python) based projects only need execution permissions for your run
# file executable to run. The executable for project 6 should be called Assembler.
# Obviously, your project may be more complicated and require a different makefile.
# IMPORTANT 1: For this file to run when you call "make", rename it from "Makefile-script" to "Makefile".
# IMPORTANT 2: If your project requires more than simply setting execution permissions, define rules
#               accordingly.

# **** How are rules defined? ****
# The following line is a rule declaration:
# all:
#     chmod a+x Assembler

# A makefile rule is a list of prerequisites (other rules that need to be run before this rule) and commands
# that are run one after the other. The "all" rule is what runs when you call "make".
# In this example, all it does is grant execution permissions for your run time executable, so your project
# will be able to run on the graders' computers. In this case, the "all" rule has no preqrequisites.

# A general rule looks like this:
# rule_name: prerequisite1 prerequisite2 prerequisite3 prerequisite4 ...
#    command1
#    command2
#    command3
#    ...
# Where each preqrequisite is a rule name, and each command is a command-line command (for example chmod,
# javac, echo, etc').

# **** Beginning of the actual Makefile ****
all:
    chmod a+x Assembler
```

7 Parser.py

```python
################################################
##   FILE : Parser.py                         ##
##   WRITER : muaz.abdeen, 300575297          ##
##   EXERCISE : nand2tetris project06 2020A   ##
################################################


class Parser:
    """
    Class that parses a valid assembly language command into its components
    """

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

    NOT_FOUND = -1
    A_COMMAND = 0
    C_COMMAND = 1
    L_COMMAND = 2

    ###################
    ##  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 (.asm) 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]
        # remove all whitespace characters (space, tab, newline, ...)
        self.command = ''.join(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 command:
        (1) A_COMMAND for @Xxx where Xxx is either a symbol or a decimal number
        (2) C_COMMAND for dest=comp;jump
        (2) L_COMMAND (actually, pseudo-command) for (Xxx) where Xxx is a symbol.
        :return: the current command type
        :rtype: int
        """
        if self.command[0] == '@':
            return self.A_COMMAND
        elif self.command[0] == '(' and self.command[-1] == ')':
            return self.L_COMMAND
        else:
            return self.C_COMMAND

    def symbol(self):
        """
        Returns the symbol or decimal Xxx of the current command @Xxx or (Xxx).
        Should be called only when commandType() is A_COMMAND or L_COMMAND.
        :return: the symbol or decimal of the current command
        :rtype: str
        """
        if self.commandType() == self.L_COMMAND:
            self.command = self.command[:-1]  # remove the left parentheses
        return self.command[1:]

    def dest(self):
        """
        Returns the dest mnemonic in the current C-command (8 possibilities).
        Should be called only when commandType() is C_COMMAND.
        :return: the dest mnemonic in the current C-command
        :rtype: str
        """
        assign_op_idx = self.command.find('=')
        if assign_op_idx == self.NOT_FOUND:
            return ''
        return self.command[:assign_op_idx]

    def comp(self):
        """
        Returns the comp mnemonic in the current C-command (28 possibilities).
        Should be called only when commandType() is C_COMMAND.
        :return: the comp mnemonic in the current C-command
        :rtype: str
        """
        assign_op_idx = self.command.find('=')
        semicolon_idx = self.command.find(';')
        if assign_op_idx == self.NOT_FOUND:
            if semicolon_idx == self.NOT_FOUND:  # no '=' or ';' (i.e: D)
                return ''
            return self.command[:semicolon_idx]  # no '=' (i.e: D;JMP)
        else:
            if semicolon_idx == self.NOT_FOUND:  # no ';' (i.e: D=M)
                return self.command[assign_op_idx + 1:]
            return self.command[assign_op_idx + 1:semicolon_idx]  # '=' and ';' (i.e: M=D;JMP)
    def jump(self):
        """
        Returns the jump mnemonic in the current C-command (8 possibilities).
        Should be called only when commandType() is C_COMMAND.
        :return: the jump mnemonic in the current C-command
        :rtype: str
        """
        semicolon_idx = self.command.find(';')
        if semicolon_idx == self.NOT_FOUND:
            return ''
        return self.command[semicolon_idx + 1:]

    # End of Parser class
```

8 SymbolTable.py

```python
##################################################
##   FILE : SymbolTable.py                     ##
##   WRITER : muaz.abdeen, 300575297           ##
##   EXERCISE : nand2tetris project06 2020A    ##
##################################################

class SymbolTable:
    """
    Class that defines a symbol table for the hack assembler.
    """

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

    def __init__(self):
        """
        initializes the predefined symbols of the assembly language
        """
        self.symbols = {
            'SP': 0, 'LCL': 1, 'ARG': 2, 'THIS': 3, 'THAT': 4,
            'R0': 0, 'R1': 1, 'R2': 2, 'R3': 3, 'R4': 4, 'R5': 5, 'R6': 6, 'R7': 7,
            'R8': 8, 'R9': 9, 'R10': 10, 'R11': 11, 'R12': 12, 'R13': 13, 'R14': 14, 'R15': 15,
            'SCREEN': 16384, 'KBD': 24576
        }

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

    def addEntry(self, symbol, address):
        """
        Adds the pair (symbol, address) to the table.
        :param symbol: the symbol to add
        :type symbol: str
        :param address: the address the symbol represents
        :type address: int
        :return: None
        """
        self.symbols[symbol] = address

    def contains(self, symbol):
        """
        Does the symbol table contain the given symbol?
        :param symbol: a given symbol to check
        :type symbol: str
        :return: True if the table contains the symbol, False else.
        :rtype: bool
        """
        return symbol in self.symbols

    def getAddress(self, symbol):
        """
        Returns the address associated with the symbol.
        :param symbol: a given symbol to get its address
        :type symbol: str
        :return: the address associated with the symbol.
        :rtype: int
        """
        return self.symbols[symbol]

    # End of SymbolTable class
```