Contents

1 Basic Test Results

```
********** TESTING FOLDER STRUCTURE START **********
Checking your submission for presence of invalid (non-ASCII) characters...
No invalid characters found.
Submission logins are: linorcohen
Is this OK?
**********  TESTING FOLDER STRUCTURE END  **********

********** PROJECT TEST START **********
Running 'make'.
'make' ran successfully.
Testing.

Running your program with command: './Assembler Add.asm'.
diff succeeded on the test.

Running your program with command: './Assembler Max.asm'.
diff succeeded on the test.

Running your program with command: './Assembler Rect.asm'.
diff succeeded on the test.
**********  PROJECT TEST END  **********

Note: the tests you see above are all the presubmission tests
for this project. The tests might not check all the different
parts of the project or all corner cases, so write your own
tests and use them!
```

2 AUTHORS

```
1 linorcohen
2 Partner 1: Linor Cohen, linor.cohen@mail.huji.ac.il, 318861226
3 Remarks:
```

3 Assembler

```
#!/bin/sh
# This file only works on Unix-like operating systems, so it won't work on Windows.

## Why do we need this file?
# The purpose of this file is to run your project.
# We want our users to have a simple API to run the project.
# So, we need a "wrapper" that will hide all  details to do so,
# enabling users to simply type 'Assembler <path>' in order to use it.

## What are '#!/bin/sh' and '$*'?
# '$*' is a variable that holds all the arguments this file has received. So, if you
# run "Assembler trout mask replica", $* will hold "trout mask replica".

## What should I change in this file to make it work with my project?
# IMPORTANT: This file assumes that the main is contained in "Main.py".
#            If your main is contained elsewhere, you will need to change this.

python3 Main.py $*

# This file is part of nand2tetris, as taught in The Hebrew University, and
# was written by Aviv Yaish. It is an extension to the specifications given
# in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
# as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
# Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

4 Code.py

```
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""

from typing import Dict


class Code:
    """Translates Hack assembly language mnemonics into binary codes."""

    dest_table = {"null": "000", "M": "001", "D": "010", "DM": "011",
                  "A": "100", "AM": "101", "AD": "110", "AMD": "111",
                  "ADM": "111", "MAD": "111"}

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

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

    @staticmethod
    def dest(mnemonic: str) -> str:
        """
        Args:
            mnemonic (str): a dest mnemonic string.

        Returns:
            str: 3-bit long binary code of the given mnemonic.
        """
        return Code.__fetch_from_table(mnemonic, Code.dest_table)

    @staticmethod
    def comp(mnemonic: str) -> str:
        """
        Args:
            mnemonic (str): a comp mnemonic string.

        Returns:
            str: the binary code of the given mnemonic.
        """
        return Code.__fetch_from_table(mnemonic, Code.comp_table)

    @staticmethod
    def jump(mnemonic: str) -> str:
        """
```

```
60          Args:
61              mnemonic (str): a jump mnemonic string.
62
63          Returns:
64              str: 3-bit long binary code of the given mnemonic.
65          """
66          return Code.jump_table[mnemonic]
67
68
69      @staticmethod
70      def __fetch_from_table(mnemonic: str, table: Dict[str, str]) -> str:
71          if mnemonic not in table:
72              return table[mnemonic[::-1]]  # support reverse
73          return table[mnemonic]
```

5 Main.py

```
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""
import os
import sys
import typing
from SymbolTable import SymbolTable
from Parser import Parser
from Code import Code

INITIAL_ADDRESS = 16
ZERO_FILL = 15
NOT_FOUND = -1
LEFT_SHIFT = "<<"
RIGHT_SHIFT = ">>"
SHIFT_CODE = "101"
C_CODE = "111"
A_CODE = "0"


def assemble_file(
        input_file: typing.TextIO, output_file: typing.TextIO) -> None:
    """Assembles a single file.

    Args:
        input_file (typing.TextIO): the file to assemble.
        output_file (typing.TextIO): writes all output to this file.
    """
    # Initialization
    first_parser = Parser(input_file)
    input_file.seek(0)
    sec_parser = Parser(input_file)
    symbol_table = SymbolTable()
    address_idx = INITIAL_ADDRESS

    # First Pass
    while first_parser.has_more_commands():
        first_parser.advance()
        if first_parser.command_type() == first_parser.L_COMMAND:
            l_symbol = first_parser.symbol()
            if not symbol_table.contains(l_symbol):
                symbol_table.add_entry(l_symbol, first_parser.command_idx + 1)

    # Second Pass
    while sec_parser.has_more_commands():
        sec_parser.advance()
        # If the instruction is @ symbol
        if sec_parser.command_type() == sec_parser.A_COMMAND:
            cur_address, address_idx = get_cur_address(address_idx, sec_parser,
                                                       symbol_table)

            # Translates the symbol to its binary value
            output_file.write(
                A_CODE + bin(int(cur_address))[2:].zfill(ZERO_FILL) + '\n')

```

```
60          # If the instruction is dest =comp ; jump
61          elif sec_parser.command_type() == sec_parser.C_COMMAND:
62              output_file.write(get_full_c_command(sec_parser))
63
64
65  def get_full_c_command(sec_parser: Parser) -> str:
66      """
67      This function returns the full binary command for type C_COMMAND
68      :param sec_parser: current parser
69      :return: string represent the binary code of the current C_COMMAND
70      """
71      comp = sec_parser.comp()
72      full_command = Code.comp(comp) + Code.dest(sec_parser.dest()) + Code.jump(
73          sec_parser.jump()) + '\n'
74      if comp.find(LEFT_SHIFT) != NOT_FOUND or comp.find(
75              RIGHT_SHIFT) != NOT_FOUND:
76          return SHIFT_CODE + full_command
77      return C_CODE + full_command
78
79
80  def get_cur_address(address_idx, sec_parser, symbol_table):
81      """
82      get the current symbol address from the symbol table
83      :param address_idx: current available address
84      :param sec_parser: secondary parser
85      :param symbol_table: the symbol table to fetch from
86      :return: the symbol address, current available address
87      """
88      cur_symbol = sec_parser.symbol()
89      if not cur_symbol.isnumeric():
90          # If symbol is not in the symbol table, adds it
91          if not symbol_table.contains(cur_symbol):
92              symbol_table.add_entry(cur_symbol, address_idx)
93              address_idx += 1
94          return symbol_table.get_address(cur_symbol), address_idx
95      return cur_symbol, address_idx
96
97
98  if "__main__" == __name__:
99      # Parses the input path and calls assemble_file on each input file.
100     # This opens both the input and the output files!
101     # Both are closed automatically when the code finishes running.
102     # If the output file does not exist, it is created automatically in the
103     # correct path, using the correct filename.
104     if not len(sys.argv) == 2:
105         sys.exit("Invalid usage, please use: Assembler <input path>")
106     argument_path = os.path.abspath(sys.argv[1])
107     if os.path.isdir(argument_path):
108         files_to_assemble = [
109             os.path.join(argument_path, filename)
110             for filename in os.listdir(argument_path)]
111     else:
112         files_to_assemble = [argument_path]
113     for input_path in files_to_assemble:
114         filename, extension = os.path.splitext(input_path)
115         if extension.lower() != ".asm":
116             continue
117         output_path = filename + ".hack"
118         with open(input_path, 'r') as input_file, \
119                 open(output_path, 'w') as output_file:
120             assemble_file(input_file, output_file)
```

6 Makefile

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

## Why do we need this file?
# We want our users to have a simple API to run the project.
# So, we need a "wrapper" that will hide all  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.
# Your project may be more complicated and require a different makefile.

## What is a makefile rule?
# 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
# executable, so your project will be able to run on the graders' computers.
# In this case, the "all" rule has no preqrequisites.

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

# 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 *

# This file is part of nand2tetris, as taught in The Hebrew University, and
# was written by Aviv Yaish. It is an extension to the specifications given
# in https://www.nand2tetris.org (Shimon Schocken and Noam Nisan, 2017),
# as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
# Unported License: https://creativecommons.org/licenses/by-nc-sa/3.0/
```

7 Parser.py

```
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""
import typing
import re


class Parser:
    """Encapsulates access to the input code. Reads an assembly program
    by reading each command line-by-line, parses the current command,
    and provides convenient access to the commands components (fields
    and symbols). In addition, removes all white space and comments.
    """
    A_COMMAND = "A_COMMAND"
    C_COMMAND = "C_COMMAND"
    L_COMMAND = "L_COMMAND"
    INITIAL_VAL = -1
    COMMENT = "//"
    NULL = "null"
    EMPTY = ""
    NOT_FOUND = -1

    def __init__(self, input_file: typing.TextIO) -> None:
        """Opens the input file and gets ready to parse it.

        Args:
            input_file (typing.TextIO): input file.
        """
        self.input_lines = input_file.read().splitlines()
        self.n = self.INITIAL_VAL
        self.command_idx = self.INITIAL_VAL
        self.cur_instruction = self.EMPTY

    def has_more_commands(self) -> bool:
        """Are there more commands in the input?

        Returns:
            bool: True if there are more commands, False otherwise.
        """
        while len(self.input_lines) - 1 != self.n:
            self.n += 1
            self.cur_instruction = self.input_lines[self.n].strip(). \
                replace(" ", "")
            if self.cur_instruction != self.EMPTY and self.cur_instruction[
                                                      0:2] != self.COMMENT:
                return True
        return False

    def advance(self) -> None:
        """Reads the next command from the input and makes it the current command.
        Should be called only if has_more_commands() is true.
        """
        if self.cur_instruction[0] != "(":  # not L_COMMAND
            self.command_idx += 1
        # remove inline comments:
```

```
60          inline_comment_idx = self.cur_instruction.find(self.COMMENT)
61          if inline_comment_idx != self.NOT_FOUND:
62              self.cur_instruction = self.cur_instruction[0:inline_comment_idx]
63
64      def command_type(self) -> str:
65          """
66          Returns:
67              str: the type of the current command:
68              "A_COMMAND" for @Xxx where Xxx is either a symbol or a decimal number
69              "C_COMMAND" for dest=comp;jump
70              "L_COMMAND" (actually, pseudo-command) for (Xxx) where Xxx is a symbol
71          """
72          first_param = self.cur_instruction[0]
73          if first_param == "(":
74              return self.L_COMMAND
75          elif first_param == "@":
76              return self.A_COMMAND
77          return self.C_COMMAND
78
79      def symbol(self) -> str:
80          """
81          Returns:
82              str: the symbol or decimal Xxx of the current command @Xxx or
83              (Xxx). Should be called only when command_type() is "A_COMMAND" or
84              "L_COMMAND".
85          """
86          command_type = self.cur_instruction[0]
87          symbol = self.cur_instruction[1:]
88          if command_type == "@":  # A_COMMAND symbol
89              return symbol
90          return symbol[:-1]  # L_COMMAND symbol
91
92      def dest(self) -> str:
93          """
94          Returns:
95              str: the dest mnemonic in the current C-command. Should be called
96              only when commandType() is "C_COMMAND".
97          """
98          dest_idx = self.cur_instruction.find("=")
99          if dest_idx == self.NOT_FOUND:
100             return self.NULL
101         return self.cur_instruction[0:dest_idx]
102
103     def comp(self) -> str:
104         """
105         Returns:
106             str: the comp mnemonic in the current C-command. Should be called
107             only when commandType() is "C_COMMAND".
108         """
109         return re.split(';', re.split('=', self.cur_instruction)[-1])[0]
110
111     def jump(self) -> str:
112         """
113         Returns:
114             str: the jump mnemonic in the current C-command. Should be called
115             only when commandType() is "C_COMMAND".
116         """
117         jump_idx = self.cur_instruction.find(";")
118         if jump_idx == self.NOT_FOUND:
119             return self.NULL
120         return self.cur_instruction[jump_idx + 1:]
```

8 SymbolTable.py

```python
"""
This file is part of nand2tetris, as taught in The Hebrew University, and
was written by Aviv Yaish. It is an extension to the specifications given
[here](https://www.nand2tetris.org) (Shimon Schocken and Noam Nisan, 2017),
as allowed by the Creative Common Attribution-NonCommercial-ShareAlike 3.0
Unported [License](https://creativecommons.org/licenses/by-nc-sa/3.0/).
"""


class SymbolTable:
    """
    A symbol table that keeps a correspondence between symbolic labels and
    numeric addresses.
    """

    def __init__(self) -> None:
        """Creates a new symbol table initialized with all the predefined symbols
        and their pre-allocated RAM addresses, according to section 6.2.3 of the
        book.
        """
        self.symbol_table = {"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}

    def add_entry(self, symbol: str, address: int) -> None:
        """Adds the pair (symbol, address) to the table.

        Args:
            symbol (str): the symbol to add.
            address (int): the address corresponding to the symbol.
        """
        self.symbol_table[symbol] = address

    def contains(self, symbol: str) -> bool:
        """Does the symbol table contain the given symbol?

        Args:
            symbol (str): a symbol.

        Returns:
            bool: True if the symbol is contained, False otherwise.
        """
        return symbol in self.symbol_table

```

```
60      def get_address(self, symbol: str) -> int:
61          """Returns the address associated with the symbol.
62
63          Args:
64              symbol (str): a symbol.
65
66          Returns:
67              int: the address associated with the symbol.
68          """
69          return self.symbol_table[symbol]
```