Contents

1 Basic Test Results

```
********** FOLDER STRUCTURE TEST START **********
Extracting submission...
     Extracted zip successfully

Finding usernames...
     Submission logins are: nogafri
     Is this OK?

Checking for non-ASCII characters with the command 'grep -IHPnsr [^\x00-\x7F] <dir>' ...
     No invalid characters found.

**********  FOLDER STRUCTURE TEST END  **********


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

Finding Assembler...
     Found in the correct path.

Testing Rect...
     Testing your Assembler with command: './Assembler Rect.asm'...
     Diff succeeded on the test.

Testing Add...
     Testing your Assembler with command: './Assembler Add.asm'...
     Diff succeeded on the test.

Testing Max...
     Testing your Assembler with command: './Assembler Max.asm'...
     Diff succeeded on the test.

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


**************************************************
**********   PRESUBMISSION TESTS PASSED   **********
**************************************************

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 nogafri
2 Partner 1: Noga Friedman, noga.fri@mail.huji.ac.il, 209010479
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/).
"""


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

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

        Returns:
            str: 3-bit long binary code of the given mnemonic.
        """
        if mnemonic== "":
            return "000"
        elif mnemonic == "M":
            return "001"
        elif mnemonic == "D":
            return "010"
        elif mnemonic == "DM" or mnemonic == "MD":  # left both options due to discrepancy between lecture/instructions
            return "011"
        elif mnemonic == "A":
            return "100"
        elif mnemonic == "AM":
            return "101"
        elif mnemonic == "AD":
            return "110"
        elif mnemonic == "ADM" or mnemonic == "AMD":  # added AMD just in case
            return "111"

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

        Returns:
            str: the binary code of the given mnemonic.
        """
        if mnemonic == "0":
            return "0101010"
        elif mnemonic == "1":
            return "0111111"
        elif mnemonic == "-1":
            return "0111010"
        elif mnemonic == "D":
            return "0001100"
        elif mnemonic == "A":
            return "0110000"
        elif mnemonic == "M":
            return "1110000"
```

```
        elif mnemonic == "!D":
            return "0001101"
        elif mnemonic == "!A":
            return "0110001"
        elif mnemonic == "!M":
            return "1110001"
        elif mnemonic == "-D":
            return "0001111"
        elif mnemonic == "-A":
            return "0110011"
        elif mnemonic == "-M":
            return "1110011"
        elif mnemonic == "D+1":
            return "0011111"
        elif mnemonic == "A+1":
            return "0110111"
        elif mnemonic == "M+1":
            return "1110111"
        elif mnemonic == "D-1":
            return "0001110"
        elif mnemonic == "A-1":
            return "0110010"
        elif mnemonic == "M-1":
            return "1110010"
        elif mnemonic == "D+A":
            return "0000010"
        elif mnemonic == "D+M":
            return "1000010"
        elif mnemonic == "D-A":
            return "0010011"
        elif mnemonic == "D-M":
            return "1010011"
        elif mnemonic == "A-D":
            return "0000111"
        elif mnemonic == "M-D":
            return "1000111"
        elif mnemonic == "D&A":
            return "0000000"
        elif mnemonic == "D&M":
            return "1000000"
        elif mnemonic == "D|A":
            return "0010101"
        elif mnemonic == "D|M":
            return "1010101"
        elif mnemonic == "A<<":
            return "0100000"
        elif mnemonic == "D<<":
            return "0110000"
        elif mnemonic == "M<<":
            return "1100000"
        elif mnemonic == "A>>":
            return "0000000"
        elif mnemonic == "D>>":
            return "0010000"
        elif mnemonic == "M>>":
            return "1000000"


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

        Returns:
            str: 3-bit long binary code of the given mnemonic.
        """
        if mnemonic == "":
```

```
128             return "000"
129         elif mnemonic == "JGT":
130             return "001"
131         elif mnemonic == "JEQ":
132             return "010"
133         elif mnemonic == "JGE":
134             return "011"
135         elif mnemonic == "JLT":
136             return "100"
137         elif mnemonic == "JNE":
138             return "101"
139         elif mnemonic == "JLE":
140             return "110"
141         elif mnemonic == "JMP":
142             return "111"
```

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


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.
    """
    symbol_table = SymbolTable()
    rom_address = 0
    ram_address = 16

    # first pass: add labels to symbol table
    parser = Parser(input_file)
    while parser.has_more_commands():
        parser.advance()
        if parser.command_type() == "L_COMMAND":
            symbol_table.add_entry(parser.symbol(), rom_address)
        else:
            rom_address += 1

    # second pass: translate commands to binary and write to output file
    input_file.seek(0)  # reset file pointer to beginning of file
    parser = Parser(input_file)
    while parser.has_more_commands():
        parser.advance()
        if parser.command_type() == "A_COMMAND":  # starts with @
            symbol = parser.symbol()  # either an integer or a variable name
            if symbol.isnumeric():
                output_file.write("0" + bin(int(symbol))[2:].zfill(15) + "\n")  ## TODO check
                continue
            elif not symbol_table.contains(symbol):
                symbol_table.add_entry(symbol, ram_address)
                ram_address += 1
            output_file.write("0" + bin(symbol_table.get_address(symbol))[2:].zfill(15) + "\n")  ## TODO check
        elif parser.command_type() == "C_COMMAND" or parser.cur_line == "0;JMP":
            if ">>" in parser.cur_line or "<<" in parser.cur_line:  # shift command
                output_file.write("101" + Code.comp(parser.comp()) + Code.dest(parser.dest()) + Code.jump(parser.jump()) + "
            else:  # regualr commands
                output_file.write("111" + Code.comp(parser.comp()) + Code.dest(parser.dest()) + Code.jump(parser.jump()) + "


if "__main__" == __name__:
    # Parses the input path and calls assemble_file on each input file.
```

```
    # This opens both the input and the output files!
    # Both are closed automatically when the code finishes running.
    # If the output file does not exist, it is created automatically in the
    # correct path, using the correct filename.
    if not len(sys.argv) == 2:
        sys.exit("Invalid usage, please use: Assembler <input path>")
    argument_path = os.path.abspath(sys.argv[1])
    if os.path.isdir(argument_path):
        files_to_assemble = [
            os.path.join(argument_path, filename)
            for filename in os.listdir(argument_path)]
    else:
        files_to_assemble = [argument_path]
    for input_path in files_to_assemble:
        filename, extension = os.path.splitext(input_path)
        if extension.lower() != ".asm":
            continue
        output_path = filename + ".hack"
        with open(input_path, 'r') as input_file, \
                open(output_path, 'w') as output_file:
            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


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.
    """

    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.lines = input_file.read().splitlines()  # saves every line as an element in a list
        self.num_lines = len(self.lines)
        self.cur_line_num = -1
        self.cur_line = ""

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

        Returns:
            bool: True if there are more commands, False otherwise.
        """
        return self.cur_line_num + 1 <= self.num_lines - 1  # True if the next potential line is within the list limits

    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.
        """
        while self.has_more_commands():
            self.cur_line_num += 1
            self.cur_line = self.lines[self.cur_line_num]

            self.cur_line = self.cur_line.replace(" ", "")  # removes whitespaces from the beginning and end of the line
            self.cur_line = self.cur_line.split("//", 1)[0]  # removes everything from "//" onwards

            # if self.cur_line.startswith("//"):
            #     continue
            if self.cur_line == "":
                continue
            break  # exits function if found a valid line

    def command_type(self) -> str:
        """
        Returns:
            str: the type of the current command:
            "A_COMMAND" for @Xxx where Xxx is either a symbol or a decimal number
            "C_COMMAND" for dest=comp;jump
```

```
            "L_COMMAND" (actually, pseudo-command) for (Xxx) where Xxx is a symbol
        """
        if self.cur_line.startswith("@"):
            return "A_COMMAND"
        elif self.cur_line.startswith("M") or self.cur_line.startswith("D") or self.cur_line.startswith("A") or self.cur_lin
            return "C_COMMAND"
        elif self.cur_line.startswith("("):
            return "L_COMMAND"

    def symbol(self) -> str:
        """
        Returns:
            str: the symbol or decimal Xxx of the current command @Xxx or
            (Xxx). Should be called only when command_type() is "A_COMMAND" or
            "L_COMMAND".
        """
        if self.command_type() == "A_COMMAND":
            return str(self.cur_line[1:])
        elif self.command_type() == "L_COMMAND":
            return self.cur_line[1:-1]

    def dest(self) -> str:
        """
        Returns:
            str: the dest mnemonic in the current C-command. Should be called
            only when commandType() is "C_COMMAND".
        """
        if "=" in self.cur_line:
            return self.cur_line.split("=")[0]  # commmand is of the form dest=comp
        else:
            return ""  # no dest

    def comp(self) -> str:
        """
        Returns:
            str: the comp mnemonic in the current C-command. Should be called
            only when commandType() is "C_COMMAND".
        """
        if "=" in self.cur_line:
            return self.cur_line.split("=")[1]  # commmand is of the form dest=comp
        elif ";" in self.cur_line:
            return self.cur_line.split(";")[0]  # command is of the form comp;jump

    def jump(self) -> str:
        """
        Returns:
            str: the jump mnemonic in the current C-command. Should be called
            only when commandType() is "C_COMMAND".
        """
        if ";" in self.cur_line:
            return self.cur_line.split(";")[1]  # command is of the form comp;jump
        else:
            return ""  # no jump
```

8 SymbolTable.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/).
"""


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.table = {'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,
                      'SP': 0, 'LCL': 1, 'ARG': 2, 'THIS': 3, 'THAT': 4, 'LOOP': 4, 'STOP': 18, 'i': 16, 'sum': 17}

    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.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.table

    def get_address(self, symbol: str) -> int:
        """Returns the address associated with the symbol.

        Args:
            symbol (str): a symbol.

        Returns:
            int: the address associated with the symbol.
        """
        return self.table[symbol]
```