Contents

1 README

```
nivkeren,ransha
==============================================================================
Niv Keren, ID 201478351, niv.keren@mail.huji.ac.il
Ran Shaham, ID 203781000, ran.shaham1@mail.huji.ac.il
==============================================================================

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


Submitted Files
---------------
README        - This file.
Makefile    - An empty file. no need to compile in Python
Assembler    - A shell script that runs the python script with the given the
          asm file name (or folder) as an argument
Assembler.py    - The main program
Code.py        - Translate the Commands from assembly to machine code
Command.py    - A class represanting a command line
Parser.py    - Parses the given file to commands list
SymbolTable.py    - Manages all the variables and labels in the assembly code

Run command
-----------

./Assembler <file_name>

Remarks
-------
* Our implementation followed the design given in the lectures
* We Implement the symbol table using the dictionary data stucture in order
  to add and get variables and labels in an efficient way.
```

2 Assembler

```
#!/bin/sh

# Runs the python script with the given argument
python3 Assembler.py $*
```

3 Assembler.py

```
import Parser,Code,sys,os
from SymbolsTable import SymbolsTable
from Command import Command

# Constants:
HACK_SUFF = ".hack"
ASM_SUFF = ".asm"
W_FILE_MODE = "w"
DEF_ENCODING = "utf-8"
ASM_SUFF_LEN = len(ASM_SUFF)

""" The assembler main file.
"""
def parse_asm_file(file_name):
    """ Gets the commands list using the parser and scans it twice
        first time searching for labels, second time uses the code to translate
        the A and C commands to machine code.
        Adds the machine code to a new .hack file
        Input: file_name - the .asm file needed to be translated
        Output: the translated file_name.hack file
    """
    line = 0
    symbols_table = SymbolsTable()
    hack_lines = []
    Parser.parse(file_name)
    # First pass
    for command in Parser.get_commands():
        if command.type == Command.L_COMMAND:
            symbols_table.add_label(command.content, line)
        else:
            line += 1
    # Second pass
    for command in Parser.get_commands():
        if command.type == Command.A_COMMAND:
            if not str(command.content).isnumeric():
                if not symbols_table.contains(command.content):
                    # a new variable
                    symbols_table.add_variable(command.content)
                command.content = symbols_table.get_address(command.content)
        elif command.type == Command.L_COMMAND:
            continue
        hack_lines.append(Code.code(command))

    #writes the hack file
    with open(file_name[:-ASM_SUFF_LEN] + HACK_SUFF, mode=W_FILE_MODE, encoding=DEF_ENCODING) as hack_file:
        for line in hack_lines:
            hack_file.write('%s\n' % line)

def main():
    """ runs the assembler on the given argument (Assembler.py <file_name>)
    """
    file_name = sys.argv[1]

    if os.path.isfile(file_name):
        parse_asm_file(file_name)
    elif os.path.isdir(file_name):
        os.chdir(file_name)
        for f in os.listdir():
            if f.endswith(ASM_SUFF):
```

```
60                  parse_asm_file(f)
61          os.chdir('..')
62
63  if __name__ == "__main__":
64      main()
```

4 Code.py

```
import re
from Command import Command
"""The translates the assembler commands to machine code commands
"""
# the regex that parses a c-command to its components
C_REGEX = '(?:(^[AMD]{1,3})=)?(?:([^;]+))(?:;(J\w{2}))?'

def code(command):
    """gets an assembler command and translates and returns it as machine code
       Input: command - the command to be translates
    """
    if command.type == Command.C_COMMAND:
        # matches the C command to the dest comp and jmp parts by groups
        match = re.match(C_REGEX, command.content)
        dest = match.group(1)
        comp = match.group(2)
        jmp  = match.group(3)
        # Convert None to empty string
        if not jmp:
            jmp = ''

        if not dest:
            dest = ''

        dest = dest.strip(); jmp = jmp.strip(); comp = comp.strip()
        dest = parse_dest(dest)
        comp = parse_comp(comp)
        jmp  = parse_jmp(jmp)
        return '1' + comp + dest + jmp
    elif command.type == Command.A_COMMAND:
        address = dec_to_binary(int(command.content))
        return '0' * (16 - len(address)) + address

def parse_dest(dest_str):
    """parses the dest part of the C command to machine code
       Input: dest_str - the dest in assembler
       Output: A string representing dest in machine code
    """
    result = 0
    if 'A' in dest_str:
        result = result | 4

    if 'D' in dest_str:
        result = result | 2

    if 'M' in dest_str:
        result = result | 1

    result = dec_to_binary(result)
    return '0' * (3-len(result)) + result

def parse_comp(comp_str):
    """parses the comp part of the C command to machine code
       Input: comp_str - the comp part in assembler
       Output: A string representing dest in machine code
    """
    comp_str = comp_str.strip()
    if comp_str == '0':
        return '110' + ('10' * 3)
```

```
60      elif comp_str == '1':
61          return '110' + ('1' * 6)
62      elif comp_str == '-1':
63          return '110111010'
64      elif comp_str == 'D':
65          return '110001100'
66      elif comp_str == 'A':
67          return '110110000'
68      elif comp_str == '!D':
69          return '110001101'
70      elif comp_str == '!A':
71          return '110110001'
72      elif comp_str == '-D':
73          return '110001111'
74      elif comp_str == '-A':
75          return '110110011'
76      elif comp_str == 'D+1':
77          return '110011111'
78      elif comp_str == 'A+1':
79          return '110110111'
80      elif comp_str == 'D-1':
81          return '110001110'
82      elif comp_str == 'A-1':
83          return '110110010'
84      elif comp_str == 'D+A':
85          return '110000010'
86      elif comp_str == 'D-A':
87          return '110010011'
88      elif comp_str == 'A-D':
89          return '110000111'
90      elif comp_str == 'D&A':
91          return '110000000'
92      elif comp_str == 'D|A':
93          return '110010101'
94      elif comp_str == 'M':
95          return '111110000'
96      elif comp_str == '!M':
97          return '111110001'
98      elif comp_str == '-M':
99          return '111110011'
100     elif comp_str == 'M+1':
101         return '111110111'
102     elif comp_str == 'M-1':
103         return '111110010'
104     elif comp_str == 'D+M':
105         return '111000010'
106     elif comp_str == 'D-M':
107         return '111010011'
108     elif comp_str == 'M-D':
109         return '111000111'
110     elif comp_str == 'D&M':
111         return '111000000'
112     elif comp_str == 'D|M':
113         return '111010101'
114     elif comp_str == 'D*A':
115         return '100000000'
116     elif comp_str == 'D*M':
117         return '101000000'
118     elif comp_str == 'D<<':
119         return '010110000'
120     elif comp_str == 'A<<':
121         return '010100000'
122     elif comp_str == 'M<<':
123         return '011100000'
124     elif comp_str == 'D>>':
125         return '010010000'
126     elif comp_str == 'A>>':
127         return '010000000'
```

```
128         elif comp_str == 'M>>':
129             return '011000000'
130 
131 
132     def parse_jmp(jmp_str):
133         """parses the jmp part of the C command to machine code
134            Input: jmp_str - the jmp in assembler
135            Ouetput: A string representing jmp in machine code
136          """
137         result = 0
138         # JGE or JGT
139         if 'G' in jmp_str:
140             result = result | 1
141 
142         # JLE or JLT
143         if 'L' in jmp_str:
144             result = result | 4
145 
146         # JLE ot JGE
147         if 'E' in jmp_str and 'N' not in jmp_str:
148             result = result | 2
149 
150         # JNE
151         elif 'NE' in jmp_str:
152             result = 5
153 
154         # unconditional jump
155         if jmp_str == 'JMP':
156             result = 7
157 
158         result = dec_to_binary(result)
159         return '0' * (3-len(result)) + result
160 
161     def dec_to_binary(dec):
162         """recieves a number in decimal representation and changes it to it binary representation
163         """
164         return str(bin(dec)[2:])
```

5 Command.py

```
class Command:
    """This class represents an assembly command."""

    # Constants:
    C_COMMAND = "C"
    A_COMMAND = "A"
    L_COMMAND = "L"

    # The type of the command - i.e. L for Label, A for Adrress, C for Command
    type = ''

    # The content of the command - i.e. The label (without the parentheses), the address (without @) or the
    # command.
    content = ''

    def __init__(self, type, content):
        """ Basic Constructor, Initializes the command variables
        Input type - the type of the command (L,A or C)
           content - the command content
        """
        self.type = type
        self.content = content
```

6 Makefile

```
# --- Empty Makefile ---
all:
```

7 Parser.py

```
import os
from Command import Command

""" The parser module for the assembler.
"""

# Constants.
COMMENT_PREFIX = '//'
READ_ONLY = 'r'
DEF_ENCODING = 'utf-8'
EMPTY_LINE = ''
A_COMMAND_PREFIX = '@'
L_COMMAND_PREFIX = '('
content = []

def parse(file_name):
    """ Parse a given assembly language file.
    """
    # Clean up when parsing a new file
    global content
    content = []
    current_command = None
    # Read the file and parse lines
    with open(file_name, mode=READ_ONLY, encoding=DEF_ENCODING) as asm_file:
        for line in asm_file:
            # Ignore whitespace & comments in the start and end of the line
            found_comment = line.find(COMMENT_PREFIX)
            if found_comment != -1:
                line = line[:found_comment]

            line = line.replace(" ", "").strip()
            if line.isspace() or line == EMPTY_LINE:
                continue
            # Determine whether current line is A/L/C Command (L for Label)
            elif line.startswith(A_COMMAND_PREFIX):
                current_command = Command(Command.A_COMMAND, line[1:])
            elif line.startswith(L_COMMAND_PREFIX):
                current_command = Command(Command.L_COMMAND, line[1:-1])
            else:
                current_command = Command(Command.C_COMMAND, line)

            # Add the created command to the content list
            content.append(current_command)

        # For loop ends here.

    # File is closed here

def get_commands():
    """ Get all commands in a parsed file - use this after running the parse function.
    This is a generator, thus running 'for command in get_commands()' will yield
    all commands in the parsed file in the correct order.
    """
    for command in content:
        yield command
```

8 SymbolsTable.py

```
class SymbolsTable:
    'A wrapper for a dictionary that holds all symbols in an asm file'
    symbols = {}
    var_num = 16

    def __init__(self):
        """ Constructor for the symbols table object.
        Adds all predefined symbols upon creation.
        """
        for i in range(16):
            self.symbols['R' + str(i)] = i
        self.symbols['SP'] = 0
        self.symbols['LCL'] = 1
        self.symbols['ARG'] = 2
        self.symbols['THIS'] = 3
        self.symbols['THAT'] = 4
        self.symbols['SCREEN'] = 16384
        self.symbols['KBD'] = 24576

    def add_variable(self, var_str):
        """ Adds a variable to the symbols table and assigns an address for it, starting from 16.
        """
        self.symbols[var_str] = self.var_num
        self.var_num += 1

    def add_label(self, label_str, label_num):
        """ Adds a label to the symbols table with a given address
        """
        self.symbols[label_str] = label_num

    def contains(self, symbol):
        """ Checks whether a given symbol exists in this symbols table
        """
        return symbol in self.symbols

    def get_address(self, symbol):
        """ Returns the address of a given symbol if it exists in the table, None otherwise.
        """
        return self.symbols[symbol]
```