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 command: './VMtranslator BasicTest.vm'
BasicTest.asm: passed the test
Running command: './VMtranslator PointerTest.vm'
PointerTest.asm: passed the test
Running command: './VMtranslator SimpleAdd.vm'
SimpleAdd.asm: passed the test
Running command: './VMtranslator StackTest.vm'
StackTest.asm: passed the test
Running command: './VMtranslator tst/SimpleAdd' where SimpleAdd is a directory.
SimpleAdd.asm: passed 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 CodeWriter.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 textwrap
import os


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

    STATIC_ADDR = 16
    TEMP_ADDR = 5

    def __init__(self, output_stream: typing.TextIO) -> None:
        """Initializes the CodeWriter.

        Args:
            output_stream (typing.TextIO): output stream.
        """
        self.output_file = output_stream
        self.filename = ""
        self.counter = 0
        self.segment_table = {"local": "LCL", "argument": "ARG",
                              "this": "THIS", "that": "THAT",
                              "static": self.STATIC_ADDR,
                              "temp": self.TEMP_ADDR}

        self.jmp_table = {"eq": "JEQ", "gt": "JGT", "lt": "JLT"}

        self.operator_table = {"add": "+", "sub": "-", "neg": "-", "not": "!",
                               "shiftright": ">>", "shiftleft": "<<",
                               "and": "&", "or": "|"}

    def __write_command_comment(self, command: str) -> None:
        """
        write a command comment before its assembly code
        :param command: a command
        """
        self.output_file.write("// " + command)

    def set_file_name(self, filename: str) -> None:
        """Informs the code writer that the translation of a new VM file is
        started.

        Args:
            filename (str): The name of the VM file.
        """
        self.filename, input_extension = \
            os.path.splitext(os.path.basename(filename))

    def write_arithmetic(self, command: str) -> None:
        """Writes assembly code that is the translation of the given
        arithmetic command. For the commands eq, lt, gt, you should correctly
        compare between all numbers our computer supports, and we define the
```

```
        value "true" to be -1, and "false" to be 0.

        Args:
            command (str): an arithmetic command.
        """
        self.__write_command_comment(command)
        text = ""
        if command in {"sub", "add"}:
            text = self.__sub_add_commands(command)
        elif command in {"neg", "not"}:
            text = self.__neg_not_commands(command)
        elif command in {"eq", "lt", "gt"}:
            text = self.__boolean_commands(command)
        elif command in {"shiftleft", "shiftright"}:
            text = self.__shift_commands(command)
        elif command in {"and", "or"}:
            text = self.__and_or_commands(command)
        self.output_file.write(textwrap.dedent(text))

    def __sub_add_commands(self, command: str) -> str:
        """
        returns the assembly code for sub or add VM command.
        :param command: (str) an sub or add command.
        :return: assembly code translation of the command
        """
        return """
        @SP
        M=M-1
        A=M
        D=M
        A=A-1
        D=M{operator}D
        M=D
        """.format(operator=self.operator_table[command])

    def __neg_not_commands(self, command: str) -> str:
        """
        returns the assembly code for neg or not VM command.
        :param command: (str) an neg or not command.
        :return: assembly code translation of the command
        """
        return """
        @SP
        A=M-1
        M={operator}M
        """.format(operator=self.operator_table[command])

    def __boolean_commands(self, command: str) -> str:
        """
        returns the assembly code for lt,gt or eq VM command.
        :param command: (str) an lt,gt or eq command.
        :return: assembly code translation of the command
        """
        self.counter += 1
        return """
        @SP
        M=M-1
        A=M
        D=M
        @NEG_{i}
        D;JLT
        @SP
        A=M-1
        D=M
        @POS_NEG_{i}
        D;JLT
        @SAME_SIGN_{i}
        0;JMP
```

```
        (NEG_{i})
        @SP
        A=M-1
        D=M
        @SAME_SIGN_{i}
        D;JLT
        D=1
        @CHECK_COMMAND_{i}
        0;JMP
        (POS_NEG_{i})
        D=-1
        @CHECK_COMMAND_{i}
        0;JMP
        (SAME_SIGN_{i})
        @SP
        A=M
        D=M
        @SP
        A=M-1
        D=M-D
        (CHECK_COMMAND_{i})
        @TRUE_{command}_{i}
        D;{command_jmp}
        @SP
        A=M-1
        M=0
        @{command}_{i}
        0;JMP
        (TRUE_{command}_{i})
        @SP
        A=M-1
        M=-1
        ({command}_{i})
        """.format(sub=self.__sub_add_commands("sub"), command=command.upper(),
                   command_jmp=self.jmp_table[command], i=self.counter)

    def __shift_commands(self, command: str) -> str:
        """
        returns the assembly code for shiftleft or shiftright VM command.
        :param command: (str) an shiftleft or shiftright command.
        :return: assembly code translation of the command
        """
        return """
        @SP
        A=M-1
        M=M{operator}
        """.format(operator=self.operator_table[command])

    def __and_or_commands(self, command: str) -> str:
        """
        returns the assembly code for and or or VM command.
        :param command: (str) an and or or command.
        :return: assembly code translation of the command
        """
        return """
        @SP
        M=M-1
        A=M
        D=M
        A=A-1
        M=D{operator}M
        """.format(operator=self.operator_table[command])

    def write_push_pop(self, command: str, segment: str, index: int) -> None:
        """Writes assembly code that is the translation of the given
        command, where command is either C_PUSH or C_POP.

        Args:
```

```
196             command (str): "C_PUSH" or "C_POP".
197             segment (str): the memory segment to operate on.
198             index (int): the index in the memory segment.
199         """
200         self.__write_command_comment(
201             f"{command[2:].lower()} {segment} {index}")
202         text = ""
203         if command == "C_PUSH":
204             text = self.__get_push_command(segment, index)
205         elif command == "C_POP":
206             text = self.__get_pop_command(segment, index)
207         self.output_file.write(textwrap.dedent(text))
208 
209     def __get_push_command(self, segment: str, index: int) -> str:
210         """
211         returns the assembly code for the given push command
212         :param segment: the memory segment to operate on.
213         :param index: the index in the memory segment.
214         :return: assembly code translation of the command
215         """
216         if segment in {"local", "argument", "this", "that", "temp"}:
217             return self.__lcl_arg_this_that_temp_push(segment, index)
218         elif segment == "static":
219             return self.__static_push(index)
220         elif segment == "constant":
221             return self.__constent_push(index)
222         elif segment == "pointer":
223             return self.__pointer_push(index)
224 
225     def __get_pop_command(self, segment: str, index: int) -> str:
226         """
227         returns the assembly code for the given pop command
228         :param segment: the memory segment to operate on.
229         :param index: the index in the memory segment.
230         :return: assembly code translation of the command
231         """
232         if segment in {"local", "argument", "this", "that", "temp"}:
233             return self.__lcl_arg_this_that_temp_pop(segment, index)
234         elif segment == "static":
235             return self.__static_pop(index)
236         elif segment == "pointer":
237             return self.__pointer_pop(index)
238 
239     def __lcl_arg_this_that_temp_push(self, segment: str, index: int) -> str:
240         """
241         returns the assembly code for push local, argument, this, that,
242         temp VM command.
243         :param segment: the memory segment to operate on.
244         :param index:  the index in the memory segment.
245         :return: assembly code translation of the command
246         """
247         return """
248         @{segmentPointer}
249         D={is_temp}
250         @{i}
251         A=D+A
252         D=M
253         @SP
254         A=M
255         M=D
256         @SP
257         M=M+1
258         """.format(segmentPointer=self.segment_table[segment], i=index,
259                    is_temp=(lambda x: "A" if x == "temp" else "M")(segment))
260 
261     def __lcl_arg_this_that_temp_pop(self, segment: str, index: int) -> str:
262         """
263         returns the assembly code for pop local, argument, this, that,
```

```
264             temp VM command.
265             :param segment: the memory segment to operate on.
266             :param index:  the index in the memory segment.
267             :return: assembly code translation of the command
268             """
269             return """
270             @{segmentPointer}
271             D={is_temp}
272             @{i}
273             D=D+A
274             @R13
275             M=D
276             @SP
277             M=M-1
278             A=M
279             D=M
280             @R13
281             A=M
282             M=D
283             """.format(segmentPointer=self.segment_table[segment], i=index,
284                        is_temp=(lambda x: "A" if x == "temp" else "M")(segment))
285
286         def __static_push(self, index: int) -> str:
287             """
288             returns the assembly code for push static VM command.
289             :param index:  the index in the memory segment.
290             :return: assembly code translation of the command
291             """
292             return """
293             @{file_name}.{i}
294             D=M
295             @SP
296             A=M
297             M=D
298             @SP
299             M=M+1
300             """.format(file_name=self.filename, i=index)
301
302         def __static_pop(self, index: int) -> str:
303             """
304             returns the assembly code for pop static VM command.
305             :param index: the index in the memory segment.
306             :return: assembly code translation of the command
307             """
308             return """
309             @SP
310             M=M-1
311             A=M
312             D=M
313             @{file_name}.{i}
314             M=D
315             """.format(file_name=self.filename, i=index)
316
317         def __constent_push(self, index: int) -> str:
318             """
319             returns the assembly code for push constant VM command.
320             :param index:  the index in the memory segment.
321             :return: assembly code translation of the command
322             """
323             return """
324             @{i}
325             D=A
326             @SP
327             A=M
328             M=D
329             @SP
330             M=M+1
331             """.format(i=index)
```

```
332
333         def __pointer_push(self, index: int) -> str:
334             """
335             returns the assembly code for push pointer (0/1 == THIS/THAT)
336             VM command.
337             :param index: the index in the memory segment.
338             :return: assembly code translation of the command
339             """
340             return """
341             @THIS
342             D=A
343             @{i}
344             A=D+A
345             D=M
346             @SP
347             A=M
348             M=D
349             @SP
350             M=M+1
351             """.format(i=index)
352
353         def __pointer_pop(self, index: int) -> str:
354             """
355             returns the assembly code for pop pointer (0/1 == THIS/THAT)
356             VM command.
357             :param index: the index in the memory segment.
358             :return: assembly code translation of the command
359             """
360             return """
361             @THIS
362             D=A
363             @{i}
364             D=D+A
365             @R13
366             M=D
367             @SP
368             M=M-1
369             A=M
370             D=M
371             @R13
372             A=M
373             M=D
374             """.format(i=index)
375
376         def write_label(self, label: str) -> None:
377             """Writes assembly code that affects the label command.
378             Let "Xxx.foo" be a function within the file Xxx.vm. The handling of
379             each "label bar" command within "Xxx.foo" generates and injects the symbol
380             "Xxx.foo$bar" into the assembly code stream.
381             When translating "goto bar" and "if-goto bar" commands within "foo",
382             the label "Xxx.foo$bar" must be used instead of "bar".
383
384             Args:
385                 label (str): the label to write.
386             """
387             # This is irrelevant for project 7,
388             # you will implement this in project 8!
389             pass
390
391         def write_goto(self, label: str) -> None:
392             """Writes assembly code that affects the goto command.
393
394             Args:
395                 label (str): the label to go to.
396             """
397             # This is irrelevant for project 7,
398             # you will implement this in project 8!
399             pass
```

```
400
401         def write_if(self, label: str) -> None:
402             """Writes assembly code that affects the if-goto command.
403
404             Args:
405                 label (str): the label to go to.
406             """
407             # This is irrelevant for project 7,
408             # you will implement this in project 8!
409             pass
410
411         def write_function(self, function_name: str, n_vars: int) -> None:
412             """Writes assembly code that affects the function command.
413             The handling of each "function Xxx.foo" command within the file Xxx.vm
414             generates and injects a symbol "Xxx.foo" into the assembly code stream,
415             that labels the entry-point to the function's code.
416             In the subsequent assembly process, the assembler translates this
417             symbol into the physical address where the function code starts.
418
419             Args:
420                 function_name (str): the name of the function.
421                 n_vars (int): the number of local variables of the function.
422             """
423             # This is irrelevant for project 7,
424             # you will implement this in project 8!
425             # The pseudo-code of "function function_name n_vars" is:
426             # (function_name)       // injects a function entry label into the code
427             # repeat n_vars times:  // n_vars = number of local variables
428             #   push constant 0     // initializes the local variables to 0
429             pass
430
431         def write_call(self, function_name: str, n_args: int) -> None:
432             """Writes assembly code that affects the call command.
433             Let "Xxx.foo" be a function within the file Xxx.vm.
434             The handling of each "call" command within Xxx.foo's code generates and
435             injects a symbol "Xxx.foo$ret.i" into the assembly code stream, where
436             "i" is a running integer (one such symbol is generated for each "call"
437             command within "Xxx.foo").
438             This symbol is used to mark the return address within the caller's
439             code. In the subsequent assembly process, the assembler translates this
440             symbol into the physical memory address of the command immediately
441             following the "call" command.
442
443             Args:
444                 function_name (str): the name of the function to call.
445                 n_args (int): the number of arguments of the function.
446             """
447             # This is irrelevant for project 7,
448             # you will implement this in project 8!
449             # The pseudo-code of "call function_name n_args" is:
450             # push return_address   // generates a label and pushes it to the stack
451             # push LCL              // saves LCL of the caller
452             # push ARG              // saves ARG of the caller
453             # push THIS             // saves THIS of the caller
454             # push THAT             // saves THAT of the caller
455             # ARG = SP-5-n_args     // repositions ARG
456             # LCL = SP              // repositions LCL
457             # goto function_name    // transfers control to the callee
458             # (return_address)      // injects the return address label into the code
459             pass
460
461         def write_return(self) -> None:
462             """Writes assembly code that affects the return command."""
463             # This is irrelevant for project 7,
464             # you will implement this in project 8!
465             # The pseudo-code of "return" is:
466             # frame = LCL                   // frame is a temporary variable
467             # return_address = *(frame-5)   // puts the return address in a temp var
```

```
468             # *ARG = pop()                  // repositions the return value for the caller
469             # SP = ARG + 1                  // repositions SP for the caller
470             # THAT = *(frame-1)             // restores THAT for the caller
471             # THIS = *(frame-2)             // restores THIS for the caller
472             # ARG = *(frame-3)              // restores ARG for the caller
473             # LCL = *(frame-4)              // restores LCL for the caller
474             # goto return_address           // go to the return address
475             pass
476
477         def close(self) -> None:
478             """
479             close the open file
480             """
481             self.output_file.close()
```

4 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 Parser import Parser
from CodeWriter import CodeWriter


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

    Args:
        input_file (typing.TextIO): the file to translate.
        output_file (typing.TextIO): writes all output to this file.
    """
    parser = Parser(input_file)
    code_writer = CodeWriter(output_file)
    code_writer.set_file_name(input_file.name)

    while parser.has_more_commands():
        parser.advance()
        c_type, arg1 = parser.command_type(), parser.arg1()
        # arithmetic
        if c_type == parser.C_ARITHMETIC:
            code_writer.write_arithmetic(arg1)
        # push pop
        elif c_type in {parser.C_POP, parser.C_PUSH}:
            code_writer.write_push_pop(c_type, arg1, parser.arg2())

    code_writer.close()


if "__main__" == __name__:
    # Parses the input path and calls translate_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: VMtranslator <input path>")
    argument_path = os.path.abspath(sys.argv[1])
    if os.path.isdir(argument_path):
        files_to_translate = [
            os.path.join(argument_path, filename)
            for filename in os.listdir(argument_path)]
        output_path = os.path.join(argument_path, os.path.basename(
            argument_path))
    else:
        files_to_translate = [argument_path]
        output_path, extension = os.path.splitext(argument_path)
    output_path += ".asm"
    with open(output_path, 'w') as output_file:
```

```
60          for input_path in files_to_translate:
61              filename, extension = os.path.splitext(input_path)
62              if extension.lower() != ".vm":
63                  continue
64              with open(input_path, 'r') as input_file:
65                  translate_file(input_file, output_file)
```

5 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 'VMtranslator <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 VMtranslator

# 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/
```

6 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:
    """
    # Parser

    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.

    ## VM Language Specification

    A .vm file is a stream of characters. If the file represents a
    valid program, it can be translated into a stream of valid assembly
    commands. VM commands may be separated by an arbitrary number of whitespace
    characters and comments, which are ignored. Comments begin with "//" and
    last until the line's end.
    The different parts of each VM command may also be separated by an
    arbitrary number of non-newline whitespace characters.

    - Arithmetic commands:
      - add, sub, and, or, eq, gt, lt
      - neg, not, shiftleft, shiftright
    - Memory segment manipulation:
      - push <segment> <number>
      - pop <segment that is not constant> <number>
      - <segment> can be any of: argument, local, static, constant, this, that,
                                 pointer, temp
    - Branching (only relevant for project 8):
      - label <label-name>
      - if-goto <label-name>
      - goto <label-name>
      - <label-name> can be any combination of non-whitespace characters.
    - Functions (only relevant for project 8):
      - call <function-name> <n-args>
      - function <function-name> <n-vars>
      - return
    """

    C_ARITHMETIC = "C_ARITHMETIC"
    C_PUSH = "C_PUSH"
    C_POP = "C_POP"
    C_LABEL = "C_LABEL"
    C_GOTO = "C_GOTO"
    C_IF = "C_IF"
    C_FUNCTION = "C_FUNCTION"
    C_RETURN = "C_RETURN"
    C_CALL = "C_CALL"
    command_table = {"push": C_PUSH, "pop": C_POP, "label": C_LABEL,
                     "goto": C_GOTO, "if": C_IF, "function": C_FUNCTION,
```

```
                    "return": C_RETURN, "call": C_CALL}
    INITIAL_VAL = -1
    COMMENT = "//"
    NULL = "null"
    EMPTY_LST = []
    EMPTY = ""
    NOT_FOUND = -1

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

        Args:
            input_file (typing.TextIO): input file.
        """
        self.input_lines = input_file.read().splitlines()
        self.n = self.INITIAL_VAL
        self.cur_command_lst = self.EMPTY_LST

    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
            cur_command = self.input_lines[self.n].strip()
            if cur_command != self.EMPTY and cur_command[
                                             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.
        Initially there is no current command.
        """
        # remove inline comments:
        cur_command = self.input_lines[self.n].strip()
        inline_comment_idx = cur_command.find(self.COMMENT)
        if inline_comment_idx != self.NOT_FOUND:
            cur_command = cur_command[0:inline_comment_idx]

        self.cur_command_lst = cur_command.split()

    def command_type(self) -> str:
        """
        Returns:
            str: the type of the current VM command.
            "C_ARITHMETIC" is returned for all arithmetic commands.
            For other commands, can return:
            "C_PUSH", "C_POP", "C_LABEL", "C_GOTO", "C_IF", "C_FUNCTION",
            "C_RETURN", "C_CALL".
        """
        if self.cur_command_lst[0] not in self.command_table:
            return self.C_ARITHMETIC
        return self.command_table[self.cur_command_lst[0]]

    def arg1(self) -> str:
        """
        Returns:
            str: the first argument of the current command. In case of
            "C_ARITHMETIC", the command itself (add, sub, etc.) is returned.
            Should not be called if the current command is "C_RETURN".
        """
        if self.command_type() == self.C_ARITHMETIC:
            return self.cur_command_lst[0]
        return self.cur_command_lst[1]
```

```
128
129     def arg2(self) -> int:
130         """
131         Returns:
132             int: the second argument of the current command. Should be
133             called only if the current command is "C_PUSH", "C_POP",
134             "C_FUNCTION" or "C_CALL".
135         """
136         return int(self.cur_command_lst[2])
```

7 VMtranslator

```
#!/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 'VMtranslator <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 "VMtranslator 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/
```