

Contents

1	Basic Test Results	2
2	README	3
3	Makefile	4
4	VMCodeWriter.py	5
5	VMParser.py	12
6	VMtranslator	14
7	VMtranslator.py	15

1 Basic Test Results

```
1 ***** TESTING FOLDER STRUCTURE START *****
2 Running test8.sh:
3 Checking your submission for presence of invalid (non-ASCII) characters...
4 No invalid characters found.
5
6 Your logins are: muaz.abdeen, is that ok?
7
8 ***** TESTING FOLDER STRUCTURE END *****
9
10 ***** PROJECT TEST START *****
11
12 Running: 'make'
13 chmod a+x VMtranslator
14 Running your program with command: './VMtranslator tst/FibonacciElement'
15 Assembler translated FibonacciElement.asm
16 Execution of FibonacciElement.asm was successful
17
18 ***** PROJECT TEST END *****
```

2 README

```
1  muaz.abdeen
2  =====
3  Muaz Abdeen, ID 300575297, muaz.abdeen@mail.huji.ac.il
4  =====
5
6                      Project 8 - VM II: Program Control
7                      -----
8
9
10 Submitted Files
11 -----
12 (1)  README           - This file.
13 (2)  VMParser.py      - The VMParser module implementation.
14 (3)  VMCodeWriter.py  - The VMCodeWriter module implementation.
15 (4)  VMtranslator.py  - The VMtranslator module implementation.
16 (5)  VMtranslator     - The run file.
17 (6)  Makefile         - The make file.
18
19
20 Remarks
21 -----
22 * ...
```

3 Makefile

```
1 all:
2     chmod a+x VMtranslator
```

4 VMCodeWriter.py

```
1 #####
2  ## FILE : VMCodeWriter.py ##
3  ## WRITER : muaz.abdeen, 300575297 ##
4  ## EXERCISE : nand2tetris project07 2020A ##
5  #####
6
7  import os
8  import VMParser
9
10
11  class VMCodeWriter:
12      """
13      Translates VM commands into Hack assembly code.
14      """
15
16      #####
17      ## MACROS & CONSTANTS ##
18      #####
19
20      #####
21      ## CONSTRUCTOR ##
22      #####
23
24      def __init__(self, output_file):
25          """
26          Opens the output file and gets ready to write it.
27          :param output_file: name of the file to write to (.asm) file
28          :type output_file: str
29          """
30          self.output_file = open(output_file, mode='w')
31          self.current_VMfile = ''
32          self.comparison_counter = 0
33          self.current_function = 'null'
34          self.call_counter = 0
35          self.writeInit()
36          # End of Constructor
37
38      #####
39      ## METHODS ##
40      #####
41
42      def setFileName(self, file_name):
43          """
44          Informs the code writer that the translation of a new VM file is started.
45          (called by the main program of the VM translator)
46          :param file_name: the new file to be translated
47          :type file_name: str
48          :return: None
49          """
50          self.current_VMfile = os.path.basename(file_name)
51
52      def writeArithmetic(self, command):
53          """
54          Writes the assembly code that is the translation of the given arithmetic command.
55          :param command: an arithmetic command.
56          :type command: str
57          :return: None
58          """
59          asm_commands = ''
```

```

60     # binary arithmetic
61     if command == 'add':
62         asm_commands = VMCodeWriter._binaryArithmetic('+')
63     elif command == 'sub':
64         asm_commands = VMCodeWriter._binaryArithmetic('-')
65     elif command == 'and':
66         asm_commands = VMCodeWriter._binaryArithmetic('&')
67     elif command == 'or':
68         asm_commands = VMCodeWriter._binaryArithmetic('|')
69
70     # unary arithmetic and logical
71     elif command == 'neg':
72         asm_commands = VMCodeWriter._unaryArithmeticOrLogical('-')
73     elif command == 'not':
74         asm_commands = VMCodeWriter._unaryArithmeticOrLogical('!')
75
76     # binary logical
77     elif command == 'eq':
78         asm_commands = self._binaryLogical('JNE')
79         # asm_commands = VMCodeWriter._binaryLogical('JEQ')
80     elif command == 'gt':
81         asm_commands = self._binaryLogical('JGE')
82     elif command == 'lt':
83         asm_commands = self._binaryLogical('JLE')
84
85     self.output_file.write(f'// {command}\n' + asm_commands)
86     # End of writeArithmetic() method
87
88     @staticmethod
89     def _binaryArithmetic(operator):
90         asm_commands = f'    @SP\n' \
91             f'        AM=M-1\n' \
92             f'        D=M\n' \
93             f'        A=A-1\n'
94         if operator == '-':
95             return asm_commands + f'        M=M-D\n'
96         # the order of commutative operations as appears in the book (D <operator> M)
97         return asm_commands + f'        M=D{operator}M\n'
98
99     @staticmethod
100     def _unaryArithmeticOrLogical(operator):
101         return f'    @SP\n' \
102             f'        A=M-1\n' \
103             f'        M={operator}M\n'
104
105     def _binaryLogical(self, jump):
106         res = (-1, 0) if jump == 'JGE' else (0, -1)
107         asm_command = f'    @SP\n' \
108             f'        AM=M-1\n' \
109             f'        D=M\n' \
110             f'        // check if y<0\n' \
111             f'        @Y_NG_{self.comparison_counter}\n' \
112             f'        D;JLT      // y<0\n' \
113             f'        // check if x<0\n' \
114             f'        @SP\n' \
115             f'        A=M-1\n' \
116             f'        D=M\n' \
117             f'        @X_NG_{self.comparison_counter}\n' \
118             f'        D;JLT      // x<0\n' \
119             f'        (SAME_SIGN_{self.comparison_counter})\n' \
120             f'        @SP\n' \
121             f'        A=M\n' \
122             f'        D=M\n' \
123             f'        A=A-1\n' \
124             f'        D=D-M\n' \
125             f'        M=-1\n' \
126             f'        @FALSE_{self.comparison_counter}\n' \
127             f'        D;{jump}\n'

```

```

128         f' @END_{self.comparison_counter}\n' \
129         f' 0;JMP\n' \
130         f' (FALSE_{self.comparison_counter})\n' \
131         f' @SP\n' \
132         f' A=M-1\n' \
133         f' M=0\n' \
134         f' @END_{self.comparison_counter}\n' \
135         f' 0;JMP\n' \
136         f' (Y_NG_{self.comparison_counter})\n' \
137         f' // check if x >= 0 \n' \
138         f' @SP\n' \
139         f' A=M-1\n' \
140         f' D=M\n' \
141         f' @SAME_SIGN_{self.comparison_counter}\n' \
142         f' D;JLT\n' \
143         f' @SP\n' \
144         f' A=M-1\n' \
145         f' M={res[0]} // y<0 , x>=0 \n' \
146         f' @END_{self.comparison_counter}\n' \
147         f' 0;JMP\n' \
148         f' (X_NG_{self.comparison_counter})\n' \
149         f' @SP\n' \
150         f' A=M-1\n' \
151         f' M={res[1]} // y>=0 , x<0 \n' \
152         f' (END_{self.comparison_counter})\n'
153     self.comparison_counter += 1
154     return asm_command
155
156 def writePushPop(self, command, segment, index):
157     """
158     Writes the assembly code that is the translation of the given command,
159     where command is either C_PUSH or C_POP.
160     :param command: a C_PUSH or C_POP command.
161     :type command: int
162     :param segment: the memory segment write to or from.
163     :type segment: str
164     :param index: the index of the memory word
165     :type index: int
166     :return: None
167     """
168     segments_map = {'local': 'LCL', 'argument': 'ARG', 'this': 'THIS', 'that': 'THAT'}
169     command_map = {VMPParser.VMPParser.C_PUSH: 'push', VMPParser.VMPParser.C_POP: 'pop'}
170     pre_comment = f'// {command_map[command]} {segment} {index}\n'
171     asm_commands = ''
172     if command == VMPParser.VMPParser.C_PUSH:
173         if segment == 'constant':
174             asm_commands = VMCodeWriter._pushConstant(index)
175         elif segment in segments_map:
176             asm_commands = VMCodeWriter._pushSegment1(segments_map[segment], index)
177         elif segment in {'temp', 'pointer', 'static'}:
178             asm_commands = self._pushSegment2(segment, index)
179
180     elif command == VMPParser.VMPParser.C_POP:
181         if segment in segments_map:
182             asm_commands = VMCodeWriter._popSegment1(segments_map[segment], index)
183         elif segment in {'temp', 'pointer', 'static'}:
184             asm_commands = self._popSegment2(segment, index)
185
186     self.output_file.write(pre_comment + asm_commands)
187     # End of writePushPop() method
188
189 @staticmethod
190 def _pushConstant(index):
191     return f' @{index}\n' \
192         f' D=A\n' \
193         f' @SP\n' \
194         f' AM=M+1\n' \
195         f' A=A-1\n' \

```

```

196         f' M=D\n'
197
198     @staticmethod
199     def _popSegment1(segment, index):
200         return f' @{{segment}}\n' \
201             f' D=M\n' \
202             f' @{{index}}\n' \
203             f' D=D+A\n' \
204             f' @R13\n' \
205             f' M=D\n' \
206             f' @SP\n' \
207             f' AM=M-1\n' \
208             f' D=M\n' \
209             f' @R13\n' \
210             f' A=M\n' \
211             f' M=D\n'
212
213     @staticmethod
214     def _pushSegment1(segment, index):
215         return f' @{{segment}}\n' \
216             f' D=M\n' \
217             f' @{{index}}\n' \
218             f' A=A+D\n' \
219             f' D=M\n' \
220             f' @SP\n' \
221             f' AM=M+1\n' \
222             f' A=A-1\n' \
223             f' M=D\n'
224
225     def _popSegment2(self, segment, index):
226         if segment == 'pointer':
227             label = 'THIS' if not index else 'THAT'
228         elif segment == 'temp':
229             label = index + 5
230         else: # static
231             label = f'{os.path.split(self.current_VMfile)[1][: -3]}.{index}'
232         return f' @SP\n' \
233             f' AM=M-1\n' \
234             f' D=M\n' \
235             f' @{{label}}\n' \
236             f' M=D\n'
237
238     def _pushSegment2(self, segment, index):
239         if segment == 'pointer':
240             label = 'THIS' if not index else 'THAT'
241         elif segment == 'temp':
242             label = index + 5
243         else: # static
244             label = f'{os.path.split(self.current_VMfile)[1][: -3]}.{index}'
245         return f' @{{label}}\n' \
246             f' D=M\n' \
247             f' @SP\n' \
248             f' AM=M+1\n' \
249             f' A=A-1\n' \
250             f' M=D\n'
251
252     def writeInit(self):
253         """
254         Writes assembly code that effects the VM initialization, also called bootstrap code.
255         This code must be placed at the beginning of the output file.
256         :return: None
257         """
258         pre_comment = '// Initializing the VM program\n'
259         asm_command = f' @256\n' \
260             f' D=A\n' \
261             f' @SP\n' \
262             f' M=D\n'
263         self.output_file.write(pre_comment + asm_command)

```



```

264         self.writeCall('Sys.init', 0)
265
266     def writeLabel(self, label):
267         """
268         Writes assembly code that effects the label command.
269         :param label: a given label
270         :type label: str
271         :return: None
272         """
273         self.output_file.write(f'({self.current_function}${label})\n')
274
275     def writeGoto(self, label):
276         """
277         Writes assembly code that effects the goto command.
278         :param label: a label with goto command
279         :type label: str
280         :return: None
281         """
282         self.output_file.write(f'    @{self.current_function}${label}\n'
283                                f'    0; JMP\n')
284
285     def writeIf(self, label):
286         """
287         Writes assembly code that effects the if-goto command.
288         :param label: a label with if-goto command
289         :type label: str
290         :return: None
291         """
292         self.output_file.write(f'// if-goto\n'
293                                f'    @SP\n'
294                                f'    M=M-1\n'
295                                f'    A=M\n'
296                                f'    D=M\n'
297                                f'    @{self.current_function}${label}\n'
298                                f'    D; JNE\n')
299
300     def writeCall(self, function_name, num_args):
301         """
302         Writes assembly code that effects the call command.
303         :param function_name: the name of the callee
304         :type function_name: str
305         :param num_args: number of the callee arguments
306         :type num_args: int
307         :return: None
308         """
309         pre_comment = f'    // call {function_name} {num_args}\n'
310         asm_command = f'    // push retAddrLabel \n' + \
311             VMCodeWriter._pushConstant(f'{function_name}$RETURN_{self.call_counter}')
312         for label in ['LCL', 'ARG', 'THIS', 'THAT']:
313             asm_command += f'    // push {label}\n' \
314                 f'    @{label}\n' \
315                 f'    D=M\n' \
316                 f'    @SP\n' \
317                 f'    M=M+1\n' \
318                 f'    A=M-1\n' \
319                 f'    M=D\n'
320         asm_command += f'    // ARG = SP-5-nArgs \n' \
321             f'    @SP\n' \
322             f'    D=M\n' \
323             f'    @5\n' \
324             f'    D=D-A\n' \
325             f'    @{num_args}\n' \
326             f'    D=D-A\n' \
327             f'    @ARG\n' \
328             f'    M=D\n' \
329             f'    // LCL = SP\n' \
330             f'    @SP\n' \
331             f'    D=M\n'

```

```

332         f' @LCL\n' \
333         f' M=D\n' \
334         f' // goto functionName\n' \
335         f' @{function_name}\n' \
336         f' 0;JMP\n' \
337         f'({function_name}$RETURN_{self.call_counter})\n'
338     self.call_counter += 1
339     self.output_file.write(pre_comment + asm_command)
340     # End of writeCall() method
341
342     def writeReturn(self):
343         """
344         Writes assembly code that effects the return command.
345         :return: None
346         """
347         pre_comment = ' // return\n'
348         asm_command = f' // endFrame (R13) = LCL\n' \
349             f' @LCL\n' \
350             f' D=M\n' \
351             f' @R13\n' \
352             f' MD=D\n' \
353             f' // retAddr (R14) = *(endFrame - 5) \n' \
354             f' @5\n' \
355             f' A=D-A\n' \
356             f' D=M\n' \
357             f' @R14\n' \
358             f' M=D\n' \
359             f' // *ARG=pop() \n' \
360             f' @SP\n' \
361             f' A=M-1\n' \
362             f' D=M\n' \
363             f' @ARG\n' \
364             f' A=M\n' \
365             f' M=D\n' \
366             f' // SP = ARG + 1\n' \
367             f' D=A\n' \
368             f' @SP\n' \
369             f' M=D+1\n'
370
371         idx = 1
372         for label in ['THAT', 'THIS', 'ARG', 'LCL']:
373             asm_command += f' // {label} = *(endFrame-{idx}) \n' \
374                 f' @R13\n' \
375                 f' D=M\n' \
376                 f' @{idx}\n' \
377                 f' A=D-A\n' \
378                 f' D=M\n' \
379                 f' @{label}\n' \
380                 f' M=D\n'
381             idx += 1
382         asm_command += f' // goto retAddr \n' \
383             f' @R14\n' \
384             f' A=M\n' \
385             f' 0;JMP\n'
386
387         self.output_file.write(pre_comment + asm_command)
388         # End of writeReturn() method
389
390     def writeFunction(self, function_name, num_locals):
391         """
392         Writes assembly code that effects the function command.
393         :param function_name: the name of the function
394         :type function_name: str
395         :param num_locals: number of local variables of the function
396         :type num_locals: int
397         :return: None
398         """
399         self.current_function = function_name
400         pre_comment = f' // function {function_name} {num_locals}\n'

```

```

400     asm_command = f'({function_name})\n'
401     if num_locals:
402         asm_command += f' @LCL\n' \
403             f' A=M\n'
404         for n in range(num_locals):
405             asm_command += f' M=0\n' \
406                 f' A=A+1\n'
407         asm_command += f' D=A\n' \
408             f' @SP\n' \
409             f' M=D\n'
410     self.output_file.write(pre_comment + asm_command)
411     # End of writeFunction() method
412
413     def close(self):
414         """
415         Closes the output file.
416         :return: None
417         """
418         self.output_file.close()
419
420     # End of VMCodeWriter class

```

5 VMParser.py

```
1 #####
2  ## FILE : VMParser.py ##
3  ## WRITER : muaz.abdeen, 300575297 ##
4  ## EXERCISE : nand2tetris projects07-08 2020A ##
5  #####
6
7
8  class VMParser:
9      """
10     Handles the parsing of a single .vm file, and encapsulates access to the input code.
11     It reads VM commands, parses them, and provides convenient access to their components.
12     In addition, it removes all white space and comments.
13     """
14
15     #####
16     ## MACROS & CONSTANTS ##
17     #####
18
19     NOT_FOUND = -1
20
21     C_ARITHMETIC = 1
22     C_PUSH = 2
23     C_POP = 3
24     C_LABEL = 4
25     C_GOTO = 5
26     C_IF = 6
27     C_FUNCTION = 7
28     C_CALL = 8
29     C_RETURN = 9
30
31     _arithmetic_commands = ['add', 'sub', 'neg', 'eq', 'gt', 'lt', 'and', 'or', 'not']
32     _type_map = {'push': C_PUSH, 'pop': C_POP,
33                  'label': C_LABEL, 'goto': C_GOTO, 'if-goto': C_IF,
34                  'function': C_FUNCTION, 'call': C_CALL, 'return': C_RETURN}
35
36     #####
37     ## CONSTRUCTOR ##
38     #####
39
40     def __init__(self, file_name):
41         """
42         Opens the input file and gets ready to parse it.
43         :param file_name: name of the file to parse (.vm) file
44         :type file_name: str
45         """
46         self.file = open(file_name, mode='r')
47         self.command = ''
48         # End of Constructor
49
50     #####
51     ## METHODS ##
52     #####
53
54     def hasMoreCommands(self):
55         """
56         Are there more commands in the input?
57         :return: True if there are more commands, False else
58         :rtype: bool
59         """
```

```

60         return self.file is not None
61
62     def advance(self):
63         """
64         Reads the next command from the input and makes it the current command.
65         Should be called only if hasMoreCommands() is true.
66         Initially there is no current command.
67         :return: None
68         """
69         self.command = self.file.readline()
70         if not self.command:
71             self.file.close()
72             self.file = None
73             return
74         # deals with comments
75         comment_idx = self.command.find('//')
76         if comment_idx != self.NOT_FOUND: # the line contains a comment
77             self.command = self.command[:comment_idx]
78         # list of command parts
79         self.command = self.command.split()
80         # blank or pure comment line
81         if not self.command:
82             self.advance()
83         # End of advance() method
84
85     def commandType(self):
86         """
87         Returns the type of the current VM command.
88         C_ARITHMETIC is returned for all the arithmetic commands.
89         :return: the current command type
90         :rtype: int
91         """
92         if self.command[0] in VMParser._arithmetic_commands:
93             return self.C_ARITHMETIC
94         return VMParser._type_map[self.command[0]]
95         # End of commandType() method
96
97     def arg1(self):
98         """
99         Returns the first argument of the current command.
100         In the case of C_ARITHMETIC, the command itself ("add", "sub", etc.) is returned.
101         Should not be called for C_RETURN.
102         :return: the first argument of the current command.
103         :rtype: str
104         """
105         assert self.commandType() != VMParser.C_RETURN
106         if self.commandType() == VMParser.C_ARITHMETIC:
107             return self.command[0]
108         return self.command[1]
109
110     def arg2(self):
111         """
112         Returns the second argument of the current command.
113         Should be called only if the current command is C_PUSH, C_POP, C_FUNCTION, or C_CALL.
114         :return: the second argument of the current command.
115         :rtype: int
116         """
117         assert len(self.command) == 3
118         return int(self.command[2])
119
120     # End of VWParser class

```

6 VMtranslator

```
1  #!/bin/sh
2  python3 VMtranslator.py $*
```

7 VMtranslator.py

```
1 #####
2  ## FILE : VMtranslator.py ##
3  ## WRITER : muaz.abdeen, 300575297 ##
4  ## EXERCISE : nand2tetris project07 2020A ##
5  #####
6
7  import VMParser
8  import VMCodeWriter
9  import sys
10 import os
11
12
13 def VMtranslator(input_file, code_writer):
14     """
15     translates the VM code file into assembly language file
16     :param input_file: VM code file
17     :type input_file: str
18     :param code_writer: the code writer
19     :type code_writer: VMCodeWriter.VMCodeWriter
20     :return: None
21     """
22     parser = VMParser.VMParser(input_file)
23     while parser.hasMoreCommands():
24         parser.advance()
25         if parser.command:
26             command_type = parser.commandType()
27             if command_type in {parser.C_POP, parser.C_PUSH}:
28                 code_writer.writePushPop(command_type, parser.arg1(), parser.arg2())
29             elif command_type == parser.C_ARITHMETIC:
30                 code_writer.writeArithmetic(parser.arg1())
31             elif command_type == parser.C_LABEL:
32                 code_writer.writeLabel(parser.arg1())
33             elif command_type == parser.C_IF:
34                 code_writer.writeIf(parser.arg1())
35             elif command_type == parser.C_GOTO:
36                 code_writer.writeGoto(parser.arg1())
37             elif command_type == parser.C_FUNCTION:
38                 code_writer.writeFunction(parser.arg1(), parser.arg2())
39             elif command_type == parser.C_CALL:
40                 code_writer.writeCall(parser.arg1(), parser.arg2())
41             elif command_type == parser.C_RETURN:
42                 code_writer.writeReturn()
43     # End of VMtranslator() function
44
45
46 def main():
47     if len(sys.argv) != 2:
48         print("Usage: VMtranslator <file.vm or path>")
49         sys.exit(-1)
50
51     program_input = sys.argv[1]
52     if os.path.isdir(program_input):
53         output_file = os.path.split(program_input)[1] + os.path.extsep + 'asm'
54         output_path = program_input + os.path.sep + output_file
55         code_writer = VMCodeWriter.VMCodeWriter(output_path)
56         for entry in os.scandir(program_input):
57             if entry.is_file() and entry.name.endswith('.vm'):
58                 code_writer.setFileName(entry.name)
59                 full_name = os.path.join(os.path.abspath(program_input), entry.name)
```

```
60         VMtranslator(full_name, code_writer)
61     code_writer.close()
62
63     elif program_input.endswith('.vm'):
64         code_writer = VMCodeWriter.VMCodeWriter(program_input.replace('vm', 'asm'))
65         code_writer.setFileName(program_input)
66         VMtranslator(program_input, code_writer)
67         code_writer.close()
68
69
70 if __name__ == '__main__':
71     main()
```