

A close-up, slightly blurred photograph of a desk. In the foreground, a white ceramic mug is on the left. To its right is a silver laptop. In front of the laptop is a spiral-bound notebook with lined paper. A black pencil lies horizontally across the notebook. A green paperclip and a green pushpin are also on the notebook. To the right of the notebook is a white ruler with black markings. The background is a light-colored desk surface.

ELEMENTS OF COMPUTING

Group assignment 4

TEAM PRESENTATION

BATCH-A TEAM-7	
GAJULA SRI VATSANKA	CB.EN.U4AIE.21010
GUNNAM HIMAMSH	CB.EN.U4AIE.21014
M.PRASANNA TEJA	CB.EN.U4AIE.21035
VIKHYAT BANSAL	CB.EN.U4AIE.21076

1. Locate the **Max.asm** and **MaxL.asm** programs in nand2tetris folder and perform the following actions. **[CO2]**
 - a. Check the correctness of both the program using the CPU emulator. Comprehend the lines of codes. (You may be asked to explain the lines of codes).
 - b. Generate machine code for the '**MaxL.asm**' program manually. Describe/comment on the translation line by line. Save the file as '**MaxL.hack**'
 - c. Develop your own assembler (using python/Java) using limited instruction set used in the program **MaxL.asm**. The assembler you developed should translate the input file '**MaxL.asm**' to '**MaxL.hack**'.
 - d. Verify the machine codes generated by your own assembler and the 'assembler' tool provided in the software suite.
 - e. Repeat (c)-(d) for **Max.asm** (Optional only, those who are good in programming can attempt it and bonus marks will be given)

GAJULA SRI VATSANKA(a,b) & VIKHYAT BANSAL(c,d,e)

a)SOLUTION

Both the MAX.asm and MAXL.asm files are located in nand2tetris project 06 folder.

I have given, which instruction is used in each line including the opcode ,destination ,comparision and jump as comments in each code.

MAX.asm	MAXL.asm
Used for symbolic program	Used without symbols
Computes max(R0,R1) and	puts the result in R2.

MAX.asm code

```
// Symbol-less version of the Max.asm program.

@0    // a instruction opcode is 0 left 15 bits represents the magnitude of 0
D=M    // c instruction destination is D & computation is M
      // D = first number

@1    // a instruction opcode is 0 left 15 bits represents the magnitude of 1
D=D-M  // c instruction destination is D & computation is D-M
      // D = first number - second number

@10   // a instruction opcode is 0 left 15 bits represents the magnitude of 10
D;JGT  // D destination code JGT is jump greater than
      // if D>0 (first is greater) goto output_first

@1    // a instruction opcode is 0 left 15 bits represents the magnitude of 1
D=M    // D destination code JGT is jump greater than
      // D = second number

@12   // a instruction opcode is 0 left 15 bits represents the magnitude of 12
0;JMP  // jump instruction
      // goto output_d

@0    // a instruction opcode is 0 left 15 bits represents the magnitude of 0
D=M    // c instruction destination is D & computation is M
      // D = first number

@2    // a instruction opcode is 0 left 15 bits represents the magnitude of 2
M=D    // c instruction destination is D & computation is M
      // M[2] = D (greatest number)

@14   // a instruction opcode is 0 left 15 bits represents the magnitude of 14
0;JMP  // jump instruction (infinity loop )
      // infinite loop
```

MAXL.asm code

```
// Computes R2 = max(R0, R1) (R0,R1,R2 refer to RAM[0],RAM[1],RAM[2])

@R0   // a instruction opcode is 0 left 15 bits represents the magnitude of 0 |
D=M    // c instruction destination is D & computation is M
      // D = first number

@R1   // a instruction opcode is 0 left 15 bits represents the magnitude of 1
D=D-M  // c instruction destination is D & computation is D-M
      // D = first number - second number

@OUTPUT_FIRST
      // a instruction opcode is 0 left 15 bits represents the magnitude of 10
D;JGT  // D destination code JGT is jump greater than
      // if D>0 (first is greater) goto output_first

@R1   // a instruction opcode is 0 left 15 bits represents the magnitude of 1
D=M    // D destination code JGT is jump greater than
      // D = second number

@OUTPUT_D
      // a instruction opcode is 0 left 15 bits represents the magnitude of 12
0;JMP  // jump instruction
      // goto output_d

(OUTPUT_FIRST)
@R0   // a instruction opcode is 0 left 15 bits represents the magnitude of 0
D=M    // c instruction destination is D & computation is M
      // D = first number

(OUTPUT_D)
@R2   // a instruction opcode is 0 left 15 bits represents the magnitude of 2
M=D    // c instruction destination is M & computation is D
      // M[2] = D (greatest number)

(INFINITE_LOOP)
@INFINITE_LOOP
      // a instruction opcode is 0 left 15 bits represents the magnitude of 14
0;JMP  // jump instruction (infinity loop )
      // infinite loop
```

Output of MAX.asm file

I have given the values of @0=8 & @1=9

CPU Emulator (2.5) - C:\Users\user\Desktop\Max (1).asm

File View Run Help

Animate: Program flow View: Screen Format: Decimal

ROM Asm

ROM	Asm
0	@0
1	D=M
2	@1
3	D=D-M
4	@10
5	D;JGT
6	@1
7	D=M
8	@12
9	O;JMP
10	@0
11	D=M
12	@2
13	M=D
14	@14
15	O;JMP
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

RAM

RAM	Value
0	8
1	9
2	9
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

Screen

D 9

ALU

D Input: 9

M/A Input: 14

ALU output: 0

PC 15

A 14

Output of MAXL.asm file

I have given the values of @0=10 & @1=5

CPU Emulator (2.5) - C:\Users\user\Desktop\MaxL (1).asm

File View Run Help

Animate: Program flow View: Screen Format: Decimal

ROM Asm

ROM	Asm
0	@0
1	D=M
2	@1
3	D=D-M
4	@10
5	D;JGT
6	@1
7	D=M
8	@12
9	O;JMP
10	@0
11	D=M
12	@2
13	M=D
14	@14
15	O;JMP
16	
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	

RAM

RAM	Value
0	10
1	5
2	10
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	0
17	0
18	0
19	0
20	0
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0

Screen

D 10

ALU

D Input: 10

M/A Input: 14

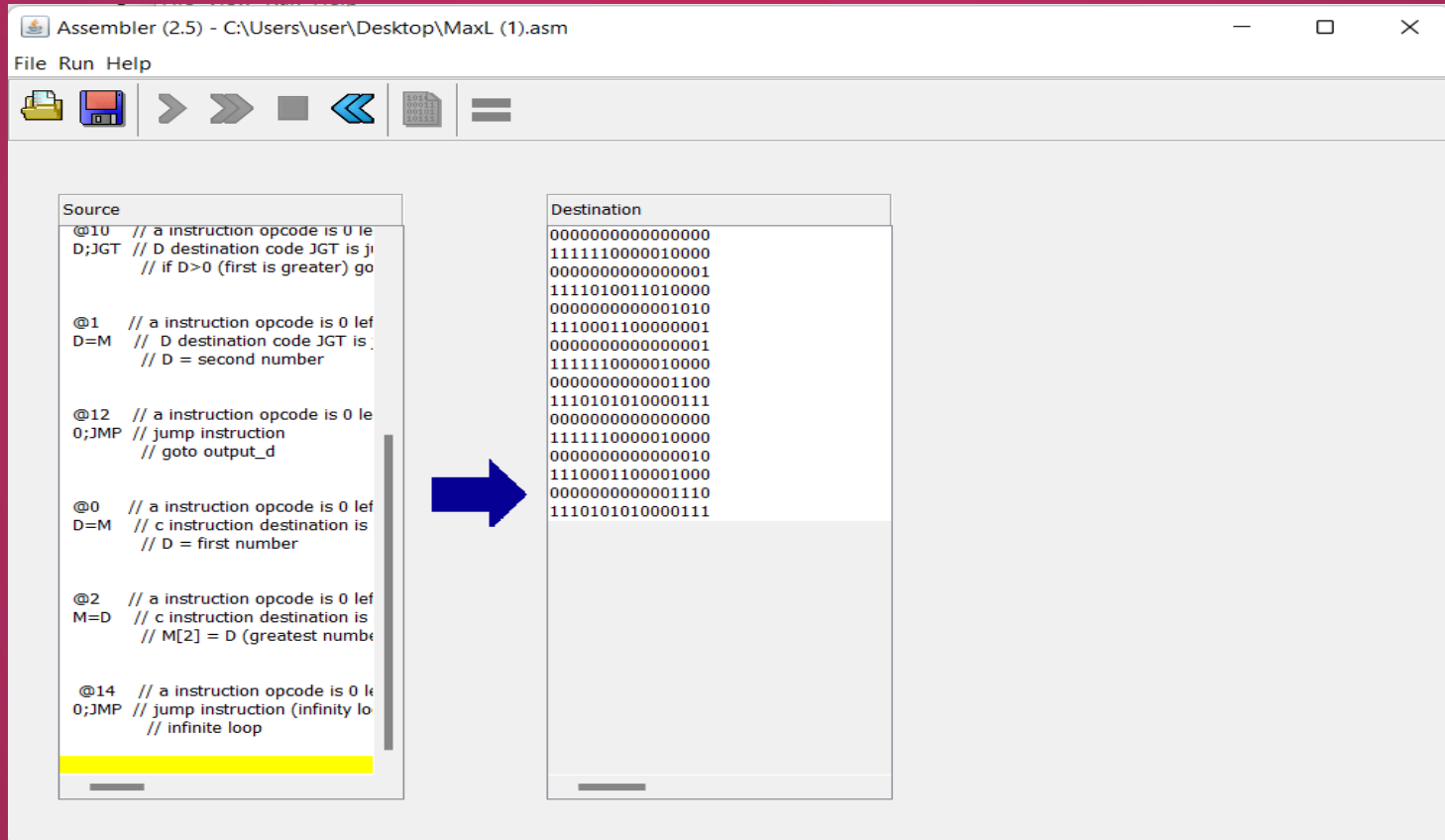
ALU output: 0

PC 14

A 14

b) SOLUTION

Open the MAXL.asm file in assembler tool and run the source code.

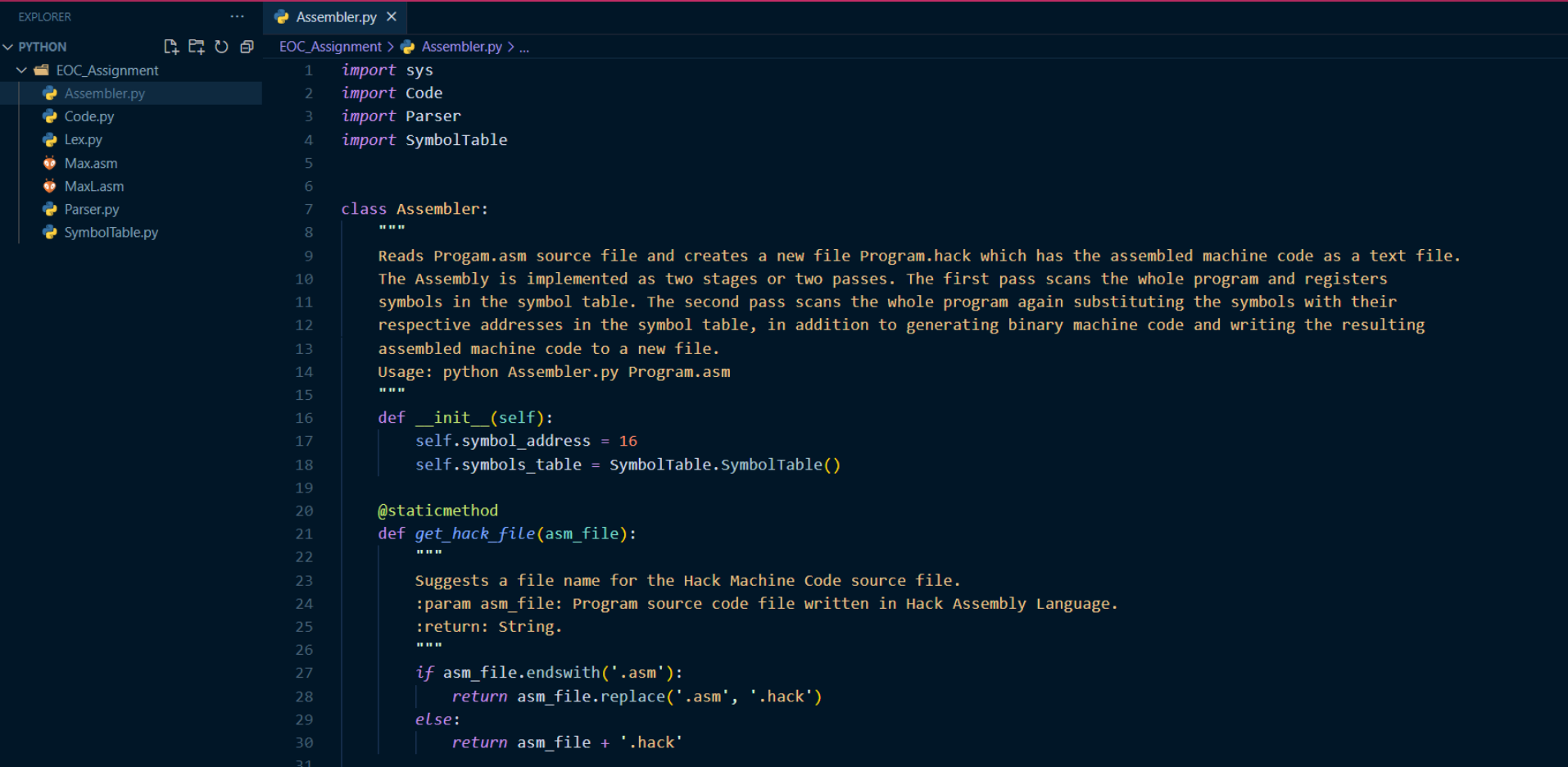


I have given, which instruction is used in each line including the opcode ,destination ,comparison and jump as comments.

```
0000000000000000 // A-instruction: @0= @valueinbinary= 0000000000000000
1111110000010000 // C-instruction: dest=D;comp=M;jump=null: 111110000010000
0000000000000001 // A-instruction: @1= @valueinbinary= 0000000000000001
1111010011010000 // C-instruction: dest=D;comp=D-M;jump=null: 1111010011010000
00000000000001010 // A-instruction: @10= @valueinbinary= 00000000000001010
1110001100000001 // C-instruction: dest=null;comp=D;jump=JGT: 1110001100000001
0000000000000001 // A-instruction: @1= @valueinbinary= 0000000000000001
1111110000010000 // C-instruction: dest=D;comp=M;jump=null: 111110000010000
00000000000001100 // A-instruction: @12= @valueinbinary= 00000000000001100
1110101010000111 // C-instruction: dest=null;comp=0;jump=JMP: 1110101010000111
0000000000000000 // A-instruction: @0= @valueinbinary= 0000000000000000
1111110000010000 // C-instruction: dest=D;comp=M;jump=null: 111110000010000
00000000000000010 // A-instruction: @2= @valueinbinary= 00000000000000010
1110001100001000 // C-instruction: dest=M;comp=D;jump=null: 1110001100001000
00000000000001110 // A-instruction: @14= @valueinbinary= 00000000000001110
1110101010000111 // C-instruction: dest=null;comp=0;jump=JMP: 1110101010000111
```


Q1 C) Complete Assembler is developed in python using different libraries and packages

Python Code:



```
1  import sys
2  import Code
3  import Parser
4  import SymbolTable
5
6
7  class Assembler:
8      """
9      Reads Program.asm source file and creates a new file Program.hack which has the assembled machine code as a text file.
10     The Assembly is implemented as two stages or two passes. The first pass scans the whole program and registers
11     symbols in the symbol table. The second pass scans the whole program again substituting the symbols with their
12     respective addresses in the symbol table, in addition to generating binary machine code and writing the resulting
13     assembled machine code to a new file.
14     Usage: python Assembler.py Program.asm
15     """
16     def __init__(self):
17         self.symbol_address = 16
18         self.symbols_table = SymbolTable.SymbolTable()
19
20     @staticmethod
21     def get_hack_file(asm_file):
22         """
23         Suggests a file name for the Hack Machine Code source file.
24         :param asm_file: Program source code file written in Hack Assembly Language.
25         :return: String.
26         """
27         if asm_file.endswith('.asm'):
28             return asm_file.replace('.asm', '.hack')
29         else:
30             return asm_file + '.hack'
31
```

```
EXPLORER
...
PYTHON
  EOC_Assignment
    Assembler.py
    Code.py
    Lex.py
    Max.asm
    MaxL.asm
    Parser.py
    SymbolTable.py

31
32 def _get_address(self, symbol):
33     """
34     Helper method. Looks-up the address of a symbol (decimal value, label or variable).
35     :param symbol: Symbol or Value.
36     :return: Address.
37     """
38     if symbol.isdigit():
39         return symbol
40     else:
41         if not self.symbols_table.contains(symbol):
42             self.symbols_table.add_entry(symbol, self.symbol_address)
43             self.symbol_address += 1
44         return self.symbols_table.get_address(symbol)
45
46 def pass_1(self, file):
47     """
48     First compilation pass: Determine memory locations of label definitions: (LABEL).
49     :param file:
50     :return:
51     """
52     parser = Parser.Parser(file)
53     curr_address = 0
54     while parser.has_more_instructions():
55         parser.advance()
56         inst_type = parser.instruction_type
57         if inst_type in [parser.A_INSTRUCTION, parser.C_INSTRUCTION]:
58             curr_address += 1
59         elif inst_type == parser.L_INSTRUCTION:
60             self.symbols_table.add_entry(parser.symbol, curr_address)
```

EXPLORER

...

Assembler.py x

PYTHON

EOC_Assignment > Assembler.py > ...

EOC_Assignment

Assembler.py

Code.py

Lex.py

Max.asm

MaxL.asm

Parser.py

SymbolTable.py

62

63

64

65

66

67

68

69

70

71

72

73

74

75

76

77

78

79

80

81

82

83

84

85

86

87

88

89

90

91

92

93

94

95

96

97

98

99

100

def pass_2(self, asm_file, hack_file):

"""

Second compilation pass: Generate hack machine code and write results to output file.

:param asm_file: The program source code file, written in Hack Assembly Language.

:param hack_file: Output file to write Hack Machine Code output to.

:return: None.

"""

parser = Parser.Parser(asm_file)

with open(hack_file, 'w', encoding='utf-8') as hack_file:

code = Code.Code()

while parser.has_more_instructions():

parser.advance()

inst_type = parser.instruction_type

if inst_type == parser.A_INSTRUCTION:

hack_file.write(code.gen_a_instruction(self._get_address(parser.symbol)) + '\n')

elif inst_type == parser.C_INSTRUCTION:

hack_file.write(code.gen_c_instruction(parser.dest, parser.comp, parser.jump) + '\n')

elif inst_type == parser.L_INSTRUCTION:

pass

def assemble(self, file):

"""

The main method. Drives the assembly process.

:param file: Program source code file, written in the Hack Assembly Language.

:return: None.

"""

self.pass_1(file)

self.pass_2(file, self.get_hack_file(file))

if __name__ == '__main__':

if len(sys.argv) != 2:

print("Usage: python Assembler.py Program.asm")

else:

asm_file = sys.argv[1]

hack_assembler = Assembler()

hack_assembler.assemble(asm_file)

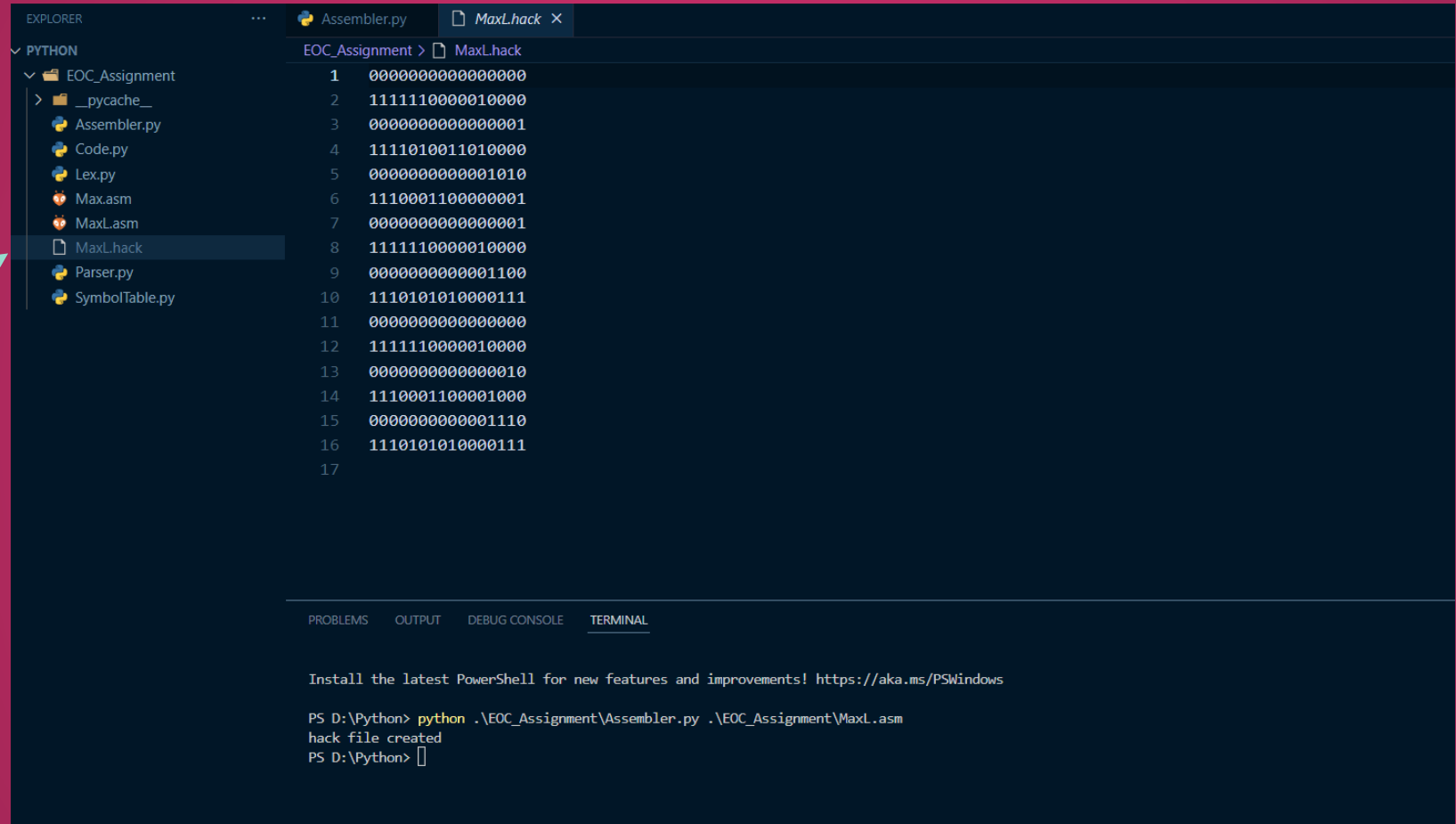
print('hack file created')

> OUTLINE

> TIMELINE

> JAVA PROJECTS

Output: Output obtained when an .asm file is passed through the code



EXPLORER

Assembly.py MaxL.hack

PYTHON

EOC_Assignment > MaxL.hack

```
1 0000000000000000
2 1111110000010000
3 0000000000000001
4 1111010011010000
5 0000000000001010
6 1110001100000001
7 0000000000000001
8 1111110000010000
9 0000000000001100
10 1110101010000111
11 0000000000000000
12 1111110000010000
13 0000000000000010
14 1110001100001000
15 0000000000001110
16 1110101010000111
17
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

Install the latest PowerShell for new features and improvements! <https://aka.ms/PSWindows>









```
PS D:\Python> python .\EOC_Assignment\Assembler.py .\EOC_Assignment\MaxL.asm
hack file created
PS D:\Python>
```

.hack file for MaxL.asm is created

Q1 D) Verification of .hack file using assembler suite tool

Assembler (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\06\max\MaxLasm

File Run Help



Source

```
// This file is part of www.nand2t
// and the book "The Elements of C
// by Nisan and Schocken, MIT Pres
// File name: projects/06/max/MaxL



// Symbol-less version of the Max.

@0
D=M
@1
D=D-M
@10
D:JGT
@1
D=M
@12
O:JMP
@0
D=M
@2
M=D
@14
O:JMP
```

Destination

Comparison

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
00000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
00000000000001110
1110101010000111
```



File created using python code



Source

```
// This file is part of www.nand2t
// and the book "The Elements of C
// by Nisan and Schocken, MIT Pres
// File name: projects/06/max/MaxL

// Symbol-less version of the Max.

@0
D=M
@1
D=D-M
@10
D;JGT
@1
D=M
@12
O;JMP
@0
D=M
@2
M=D
@14
O;JMP
```



Destination

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```



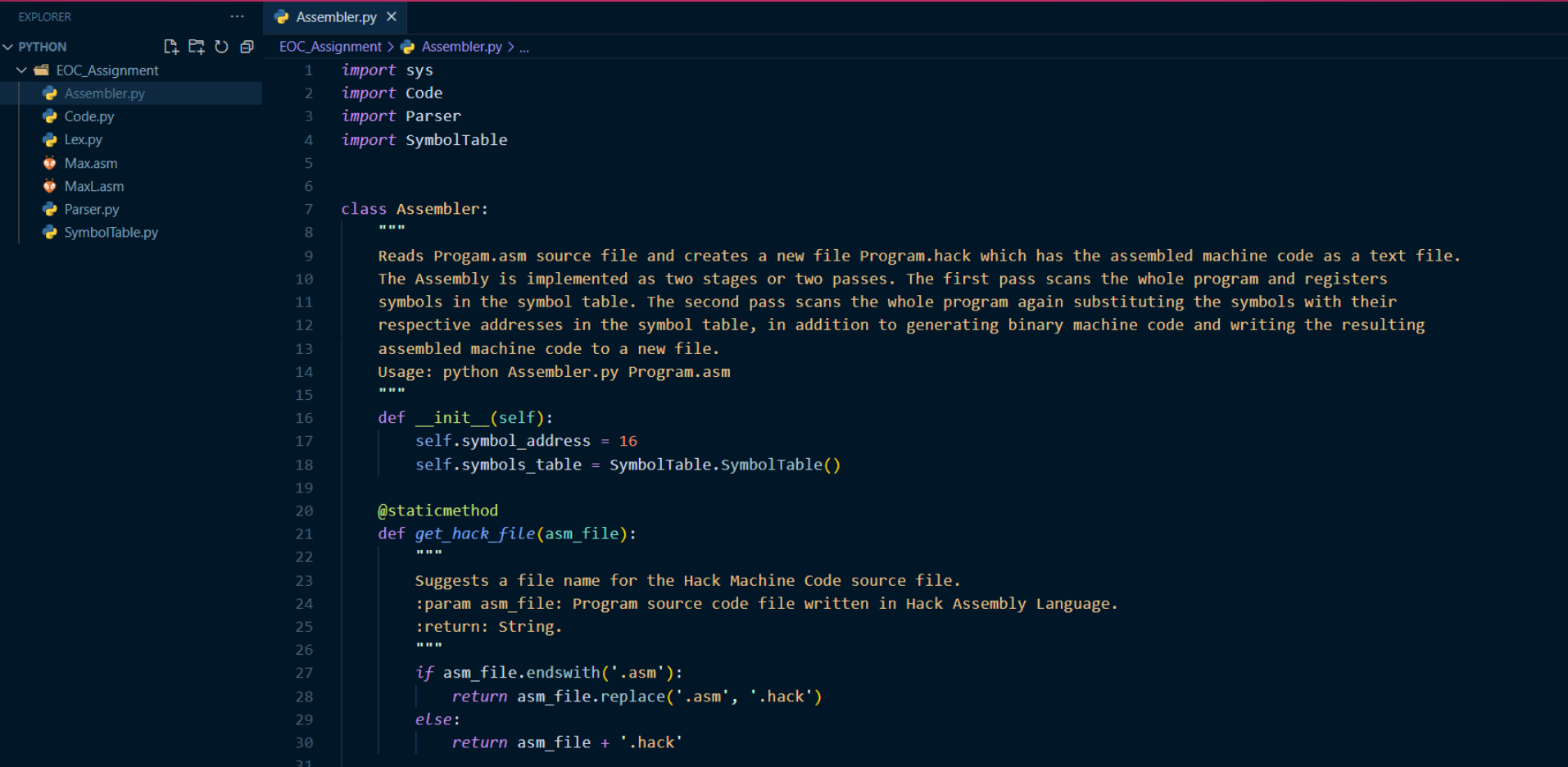
Comparison

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

File compilation & comparison succeeded

Q1 E) i) Complete Assembler is developed in python using different libraries and packages

Python Code:



```
1  import sys
2  import Code
3  import Parser
4  import SymbolTable
5
6
7  class Assembler:
8      """
9      Reads Program.asm source file and creates a new file Program.hack which has the assembled machine code as a text file.
10     The Assembly is implemented as two stages or two passes. The first pass scans the whole program and registers
11     symbols in the symbol table. The second pass scans the whole program again substituting the symbols with their
12     respective addresses in the symbol table, in addition to generating binary machine code and writing the resulting
13     assembled machine code to a new file.
14     Usage: python Assembler.py Program.asm
15     """
16     def __init__(self):
17         self.symbol_address = 16
18         self.symbols_table = SymbolTable.SymbolTable()
19
20     @staticmethod
21     def get_hack_file(asm_file):
22         """
23         Suggests a file name for the Hack Machine Code source file.
24         :param asm_file: Program source code file written in Hack Assembly Language.
25         :return: String.
26         """
27         if asm_file.endswith('.asm'):
28             return asm_file.replace('.asm', '.hack')
29         else:
30             return asm_file + '.hack'
31
```

```
EXPLORER
...
PYTHON
  EOC_Assignment
    Assembler.py
    Code.py
    Lex.py
    Max.asm
    MaxL.asm
    Parser.py
    SymbolTable.py

31
32 def _get_address(self, symbol):
33     """
34     Helper method. Looks-up the address of a symbol (decimal value, label or variable).
35     :param symbol: Symbol or Value.
36     :return: Address.
37     """
38     if symbol.isdigit():
39         return symbol
40     else:
41         if not self.symbols_table.contains(symbol):
42             self.symbols_table.add_entry(symbol, self.symbol_address)
43             self.symbol_address += 1
44         return self.symbols_table.get_address(symbol)
45
46 def pass_1(self, file):
47     """
48     First compilation pass: Determine memory locations of label definitions: (LABEL).
49     :param file:
50     :return:
51     """
52     parser = Parser.Parser(file)
53     curr_address = 0
54     while parser.has_more_instructions():
55         parser.advance()
56         inst_type = parser.instruction_type
57         if inst_type in [parser.A_INSTRUCTION, parser.C_INSTRUCTION]:
58             curr_address += 1
59         elif inst_type == parser.L_INSTRUCTION:
60             self.symbols_table.add_entry(parser.symbol, curr_address)
```


▼ EOC Assignment

 Assembler.py

 Code.py

Lex.py

 Max.asm MaxL.asm Parser.py

SymbolTable.py

```

62 def pass_2(self, asm_file, hack_file):
63     """
64     Second compilation pass: Generate hack machine code and write results to output file.
65     :param asm_file: The program source code file, written in Hack Assembly Language.
66     :param hack_file: Output file to write Hack Machine Code output to.
67     :return: None.
68     """
69     parser = Parser.Parser(asm_file)
70     with open(hack_file, 'w', encoding='utf-8') as hack_file:
71         code = Code.Code()
72         while parser.has_more_instructions():
73             parser.advance()
74             inst_type = parser.instruction_type
75             if inst_type == parser.A_INSTRUCTION:
76                 hack_file.write(code.gen_a_instruction(self._get_address(parser.symbol)) + '\n')
77             elif inst_type == parser.C_INSTRUCTION:
78                 hack_file.write(code.gen_c_instruction(parser.dest, parser.comp, parser.jmp) + '\n')
79             elif inst_type == parser.L_INSTRUCTION:
80                 pass
81
82 def assemble(self, file):
83     """
84     The main method. Drives the assembly process.
85     :param file: Program source code file, written in the Hack Assembly Language.
86     :return: None.
87     """
88     self.pass_1(file)
89     self.pass_2(file, self.get_hack_file(file))
90
91
92 if __name__ == '__main__':
93     if len(sys.argv) != 2:
94         print("Usage: python Assembler.py Program.asm")
95     else:
96         asm_file = sys.argv[1]
97
98         hack_assembler = Assembler()
99         hack_assembler.assemble(asm_file)
100         print('hack file created')

```

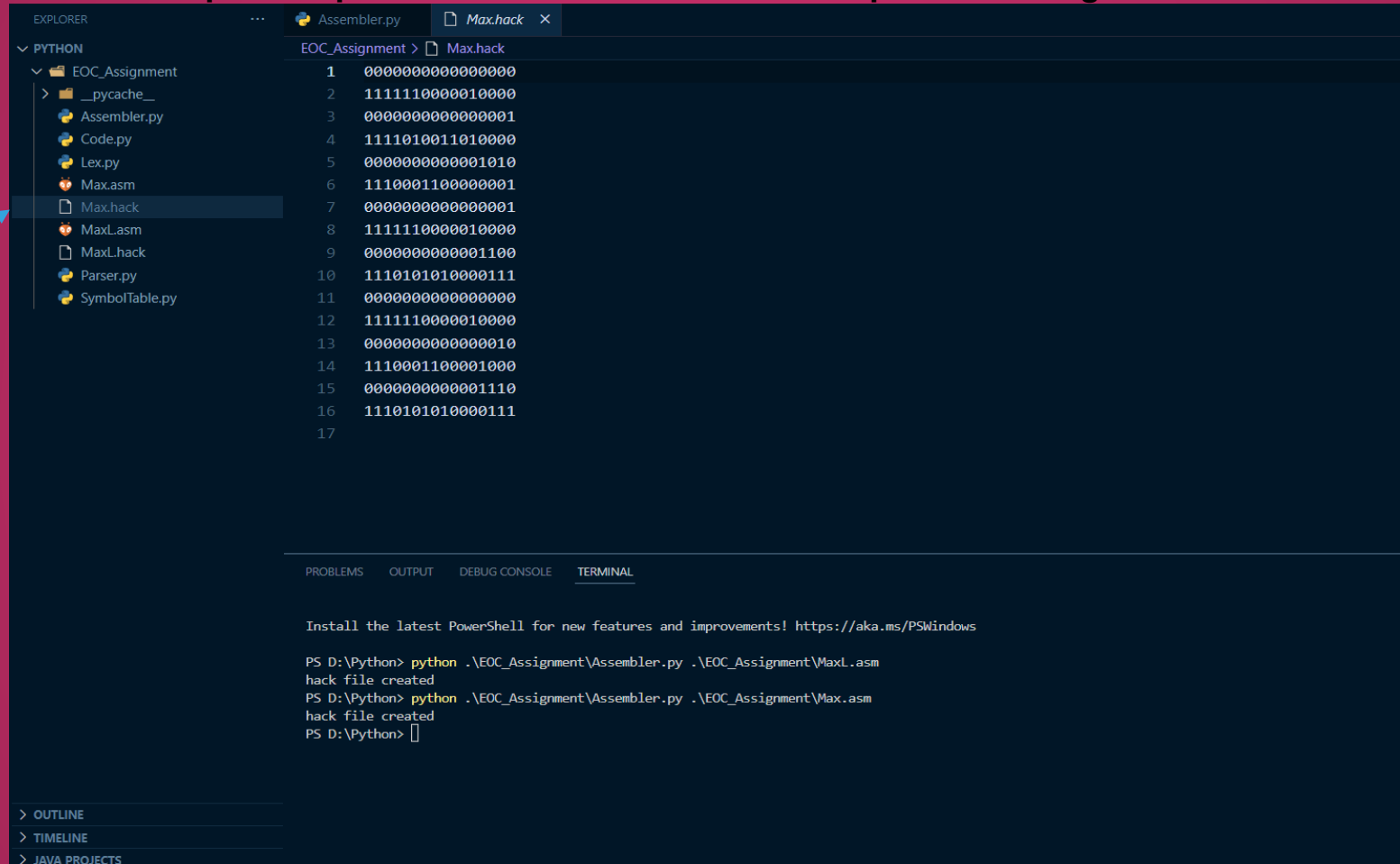
> OUTLINE

> TIMELINE

> JAVA PROJECTS

Output: Output obtained when an .asm file is passed through the code

Hack file
appears



The screenshot shows a VS Code editor with a project named 'EOC_Assignment'. The Explorer sidebar on the left lists files under the 'PYTHON' folder, including 'Max.asm' and 'Max.hack'. A blue arrow points to 'Max.hack' with the text 'Hack file appears'. The main editor area shows the content of 'Max.hack', which consists of 17 lines of binary code (0s and 1s). The bottom panel shows the 'TERMINAL' output, which includes a PowerShell prompt and the execution of a Python script that creates the 'Max.hack' file from 'Max.asm'.

```
EXPLORER
PYTHON
  EOC_Assignment
    __pycache__
    Assembler.py
    Code.py
    Lex.py
    Max.asm
    Max.hack
    MaxL.asm
    MaxL.hack
    Parser.py
    SymbolTable.py

EOC_Assignment > Max.hack
1  0000000000000000
2  1111110000010000
3  0000000000000001
4  1111010011010000
5  0000000000001010
6  1110001100000001
7  0000000000000001
8  1111110000010000
9  0000000000001100
10 1110101010000111
11 0000000000000000
12 1111110000010000
13 0000000000000010
14 1110001100001000
15 0000000000001110
16 1110101010000111
17

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL









Install the latest PowerShell for new features and improvements! https://aka.ms/PSWindows

PS D:\Python> python .\EOC_Assignment\Assembler.py .\EOC_Assignment\MaxL.asm
hack file created
PS D:\Python> python .\EOC_Assignment\Assembler.py .\EOC_Assignment\Max.asm
hack file created
PS D:\Python>
```

.hack file for Max.asm is created

Q1 E) ii) Verification of .hack file using assembler suite tool

FileRunHelp



Source

```
// This file is part of www.nand2t
// and the book "The Elements of C
// by Nisan and Schocken, MIT Pres
// File name: projects/06/max/Max.

// Computes R2 = max(R0, R1) (R0,


    @R0
    D=M          // D = first n
    @R1
    D=D-M        // D = first n
    @OUTPUT_FIRST
    D;JGT        // if D>0 (fir
    @R1
    D=M          // D = second
    @OUTPUT_D
    0;JMP        // goto output
(OUTPUT_FIRST)
    @R0
    D=M          // D = first n
(OUTPUT_D)
    @R2
    M=D          // M[2] = D (g
(INFINITE_LOOP)
    @INFINITE_LOOP
    0;JMP        // infinite lo
```

Destination

Comparison

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

File created using python code



19



Source

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press
// File name: projects/06/max/Max.asm

// Computes R2 = max(R0, R1)  (R0, R1 are inputs)

    @R0
    D=M           // D = first number
    @R1
    D=D-M         // D = first number minus second
    @OUTPUT_FIRST
    D;JGT         // if D > 0 (first > second) jump to output first
    @R1
    D=M           // D = second number
    @OUTPUT_D
    0;JMP         // goto output second
(OUTPUT_FIRST)
    @R0
    D=M           // D = first number
    @OUTPUT_D
    @R2
    M=D           // M[2] = D (store second number)
(INFINITE_LOOP)
    @INFINITE_LOOP
    0;JMP         // infinite loop
```

Destination

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

Comparison

```
0000000000000000
1111110000010000
0000000000000001
1111010011010000
0000000000001010
1110001100000001
0000000000000001
1111110000010000
0000000000001100
1110101010000111
0000000000000000
1111110000010000
0000000000000010
1110001100001000
0000000000001110
1110101010000111
```

2

2. Write an assembly language program to find the average of **N** numbers and save the file as **'average.asm'** [CO2]
 - a. Check the correctness of the program using the CPU emulator.
 - b. Generate machine code for the program manually. Describe/comment on the translation line by line. Save the file as **'average.hack'**
 - c. Compare the machine codes generated manually and the one generated by the 'assembler' tool.

GUNNAM HIMAMSH & M.PRASANNA TEJA

```

//Usage: Computes the average of n numbers
@0
D=M          // D = RAM[0] = n . we have to take any n value here
@n
M=D          // M = n

@i
M=0  // i=0

@sum
M=0  // sum=0

(LOOP)
@i
D=M  // D = i
@n
D=D-M
@SUB
D;JGT // if i>n goto SUB
@sum
D=M
@i
D=D+M
@sum
M=D  // sum = sum + i
@i
M=M+1 // i = i + 1
@LOOP
0;JMP

```

```
(SUB)
@sum
D=M
@j
M=0
(SUBA)
@0
D=D-M
@END
D;JLT
@j
M=M+1
@avg
M=D
@SUBA
D;JGT

(END)
@END
0;JMP    // Infinite loop
```

EXPLANATION

- ❑ *The above assembly code is to get average of n numbers.*
- ❑ *First, we take n in the memory, and we can keep whatever value we want in place of n in the RAM*
- ❑ *So, after storing the value in the RAM we to take the data and after that we have to take another variable, we have to store the data in that memory as this is the first variable it will be stored as @16 .*
- ❑ *We have to take another 2 variables @sum and @l for selecting the value for the loop and sum to store the final value we will give 0 for those to initially .*
- ❑ *We will start the loop by starting using @l as we progress in the loop, l value increments by 1 and by this loop and the code we use we will find the sum .*
- ❑ *The loop continues until l value is greater than n , n will be taken as our wish.*

CONTINUATION

- ❑ After that we will find the average and the formula is sum of total number of elements divided by total number of elements.
- ❑ As we do not have any operation to divide two number, we can do it by subtraction.
- ❑ In this we subtract 10 with the value in the variable sum and continue this process until we get the last positive number and this the average value of array elements.
- ❑ We will store this value in a variable avg and the program by writing an infinite loop.
- ❑ And at last, we will make the end loop so to make program ends here with an unconditional jump and hence, our values are swapped.

ROM	Asm	
0	@0	
1	D=M	
2	@16	
3	M=D	
4	@17	
5	M=0	
6	@18	
7	M=0	
8	@17	
9	D=M	
10	@16	
11	D=D-M	
12	@24	
13	D; JGT	
14	@18	
15	D=M	
16	@17	
17	D=D+M	
18	@18	
19	M=D	
20	@17	
21	M=M+1	
22	@8	
23	0; JMP	
24	@18	
25	D=M	
26	@19	
27	M=0	
28	@0	




PC

38

RAM	
0	10
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0
10	0
11	0
12	0
13	0
14	0
15	0
16	10
17	11
18	55
19	5
20	5
21	0
22	0
23	0
24	0
25	0
26	0
27	0
28	0



A

38

ROM		Asm			
15	D=M				
16	@17				
17	D=D+M				
18	@18				
19	M=D				
20	@17				
21	M=M+1				
22	@8				
23	0; JMP				
24	@18				
25	D=M				
26	@19				
27	M=0				
28	@0				
29	D=D-M				
30	@38				
31	D; JLT				
32	@19				
33	M=M+1				
34	@20				
35	M=D				
36	@28				
37	D; JGT				
38	@38				
39	0; JMP				
40					
41					
42					
43					




PC

38

RAM			
0	10		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	0		
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		
16	10		
17	11		
18	55		
19	5		
20	5		
21	0		
22	0		
23	0		
24	0		
25	0		
26	0		
27	0		
28	0		



A

38

ROM		Asm			
15		D=M			
16		@17			
17		D=D+M			
18		@18			
19		M=D			
20		@17			
21		M=M+1			
22		@8			
23		0; JMP			
24		@18			
25		D=M			
26		@19			
27		M=0			
28		@0			
29		D=D-M			
30		@38			
31		D; JLT			
32		@19			
33		M=M+1			
34		@20			
35		M=D			
36		@28			
37		D; JGT			
38		@38			
39		0; JMP			
40					
41					
42					
43					

PC

39

RAM			
0	15		
1	0		
2	0		
3	0		
4	0		
5	0		
6	0		
7	0		
8	0		
9	0		
10	0		
11	0		
12	0		
13	0		
14	0		
15	0		
16	15		
17	16		
18	120		
19	8		
20	0		
21	0		
22	0		
23	0		
24	0		
25	0		
26	0		
27	0		
28	0		

A

38

Source

```
@0
D=M
@n
M=D

@i
M=0 // i=0

@sum
M=0 // sum=0

(LOOP)
@i
D=M
@n
D=D-M
@SUB
D;JGT // if i>n goto SUB
@sum
D=M
@i
D=D+M
@sum
M=D // sum = sum + i
@i
M=M+1 // i = i + 1
@LOOP
O;JMP

(SUB)
```



Destination

```
0000000000000000
1111110000010000
0000000000010000
1110001100001000
0000000000010001
1110101010001000
0000000000010010
1110101010001000
0000000000010001
1111110000010000
0000000000010000
1111010011010000
0000000000011000
1110001100000001
0000000000010010
1111110000010000
0000000000010001
1111000010010000
0000000000010010
1110001100001000
0000000000010001
1111110111001000
0000000000001000
1110101010000111
0000000000010010
1111110000010000
0000000000010011
1110101010001000
0000000000000000
1111010011010000
```

Source

```
D=M
@i
D=D+M
@sum
M=D // sum = sum + i
@i
M=M+1 // i = i + 1
@LOOP
0;JMP

(SUB)
@sum
D=M
@j
M=0
(SUBA)
@0
D=D-M
@END
D;JLT
@j
M=M+1
@avg
M=D
@SUBA
D;JGT

(END)
@END
0;JMP // Infinite loop
```



Destination

```
00000000000010000
1111010011010000
00000000000011000
1110001100000001
00000000000010010
1111110000010000
00000000000010001
1111000010010000
00000000000010010
1110001100001000
00000000000010001
1111110111001000
0000000000001000
1110101010000111
00000000000010010
1111110000010000
00000000000010011
1110101010001000
00000000000000000
1111010011010000
00000000000100110
1110001100000100
00000000000010011
1111110111001000
00000000000010100
1110001100001000
00000000000011100
1110001100000001
000000000000100110
1110101010000111
```

```

//For A instruction the binary syntax is we represent them in 16 bit binary numbers
//For C instruction the Symbolic syntax is Dest=comp;jump and binary syntax is 1 1 1 a c1 c2 c3 c4 c5 c6 d1 d2 d3
0000000000000000 // for @0 all are 0's
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000 (Syntax for c instruction is dest = comp;jump)
0000000000001000 // it is a instruction so binary representation of 16 is represented
1110001100001000 // a = 0 ; c = 001100 ; d = 001 ; j = 000
0000000000001001 // it is a instruction so binary representation of 17 is represented
1110101010001000 // a = 0 ; c = 101010 ; d = 001 ; j = 000
0000000000001010 // it is a instruction so binary representation of 18 is represented
1110101010001000 // a = 0 ; c = 101010 ; d = 001 ; j = 000
0000000000001001 // it is a instruction so binary representation of 17 is represented
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000
0000000000001000 // it is a instruction so binary representation of 16 is represented
1111010011010000 // a = 1; c = 010011 ; d = 010 ; j = 000
00000000000011000 // it is a instruction so binary representation of 24 is represented
1110001100000001 // a = 0 ; c = 001100 ;d = 000 ; j = 001
0000000000001010 // it is a instruction so binary representation of 18 is represented
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000
0000000000001001 // it is a instruction so binary representation of 17 is represented
1111000010010000 // a = 1 ; c = 000010 ; d = 010 ; j = 000
0000000000001010 // it is a instruction so binary representation of 18 is represented
1110001100001000 // a = 0 ; c = 001100 ; d = 001 ; j = 000
0000000000001001 // it is a instruction so binary representation of 17 is represented
1111110111001000 // a = 1 ; c = 110111 ; d = 001 ; j= 000
00000000000001000 // it is a instruction so binary representation of 8 is represented
1110101010000111 // a = 0 ; c = 101010 ; d = 000 ; j = 111
0000000000001010 // it is a instruction so binary representation of 18 is represented
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000
00000000000010011 // it is a instruction so binary representation of 19 is represented
1110101010001000 // a = 0 ; c = 101010 ; d = 001 ; j = 000

```

```
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000
0000000000010011 // it is a instruction so binary representation of 19 is represented
1110101010001000 // a = 0 ; c = 101010 ; d = 001 ; j = 000
0000000000000000 // for @0 all are 0's
1111010011010000 // a = 1 ; c = 010011 ; d = 010 ; j = 000
00000000000100110 // it is a instruction so binary representation of 38 is represented
1110001100000100 // a = 0 ; c = 001100 ; d = 000 ; j = 100
0000000000010011 // it is a instruction so binary representation of 19 is represented
1111110111001000 // a = 1 ; c = 110111 ; d = 001 ; j = 000
0000000000010100 // it is a instruction so binary representation of 20 is represented
1110001100001000 // a = 0 ; c = 001100 ; d = 001 ; j = 000
0000000000011100 // it is a instruction so binary representation of 28 is represented
1110001100000001 // a = 0 ; c = 001100 ; d = 000 ; j = 001
00000000000100110 // it is a instruction so binary representation of 38 is represented
1110101010000111 // a = 0 ; c = 101010 ; d = 000 ; j = 111
```


Source

```
@0
D=M
@n
M=D

@i
M=0 // i=0

@sum
M=0 // sum=0

(LOOP)
@i
D=M
@n
D=D-M
@SUB
D;JGT // if i>n goto SUB
@sum
D=M
@i
D=D+M
@sum
M=D // sum = sum + i
@i
M=M+1 // i = i + 1
@LOOP
0;JMP

(SUB)
```

Destination

```
0000000000000000
1111110000010000
0000000000010000
1110001100001000
0000000000010001
1110101010001000
0000000000010010
1110101010001000
0000000000010001
1111110000010000
0000000000010000
1111010011010000
0000000000011000
1110001100000001
0000000000010010
1111110000010000
0000000000010001
1111000010010000
0000000000010010
1110001100001000
0000000000010001
1111110111001000
0000000000001000
1110101010000111
0000000000010010
1111110000010000
0000000000010011
1110101010001000
0000000000000000
1111010011010000
```

Comparison

```
0000000000000000
1111110000010000
0000000000010000
1110001100001000
0000000000010001
1110101010001000
0000000000010010
1110101010001000
0000000000010001
1111110000010000
0000000000010000
1111010011010000
0000000000011000
1110001100000001
0000000000010010
1111110000010000
0000000000010001
1111000010010000
0000000000010010
1110001100001000
0000000000010001
1111110111001000
0000000000001000
1110101010000111
0000000000010010
1111110000010000
0000000000010011
1110101010001000
0000000000000000
1111010011010000
```

Source

```
D=M
@i
D=D+M
@sum
M=D // sum = sum + i
@i
M=M+1 // i = i + 1
@LOOP
0;JMP

(SUB)
@sum
D=M
@j
M=0
(SUBA)
@0
D=D-M
@END
D;JLT
@j
M=M+1
@avg
M=D
@SUBA
D;JGT

(END)
@END
0;JMP // Infinite loop
```



Destination

```
00000000000010000
1111010011010000
00000000000011000
11100011000000001
00000000000010010
1111110000010000
00000000000010001
1111000010010000
00000000000010010
1110001100001000
00000000000010001
1111110111001000
0000000000001000
1110101010000111
00000000000010010
1111110000010000
00000000000010011
1110101010001000
00000000000000000
1111010011010000
000000000000100110
1110001100000100
00000000000010011
1111110111001000
00000000000010100
1110001100001000
00000000000011100
11100011000000001
000000000000100110
1110101010000111
```



Comparison

```
00000000000010000
1111010011010000
00000000000011000
11100011000000001
00000000000010010
1111110000010000
00000000000010001
1111000010010000
00000000000010010
1110001100001000
00000000000010001
1111110111001000
0000000000001000
1110101010000111
00000000000010010
1111110000010000
00000000000010011
1110101010001000
00000000000000000
1111010011010000
000000000000100110
1110001100000100
00000000000010011
1111110111001000
00000000000010100
1110001100001000
00000000000011100
11100011000000001
000000000000100110
1110101010000111
```

File compilation & comparison succeeded

THANK YOU