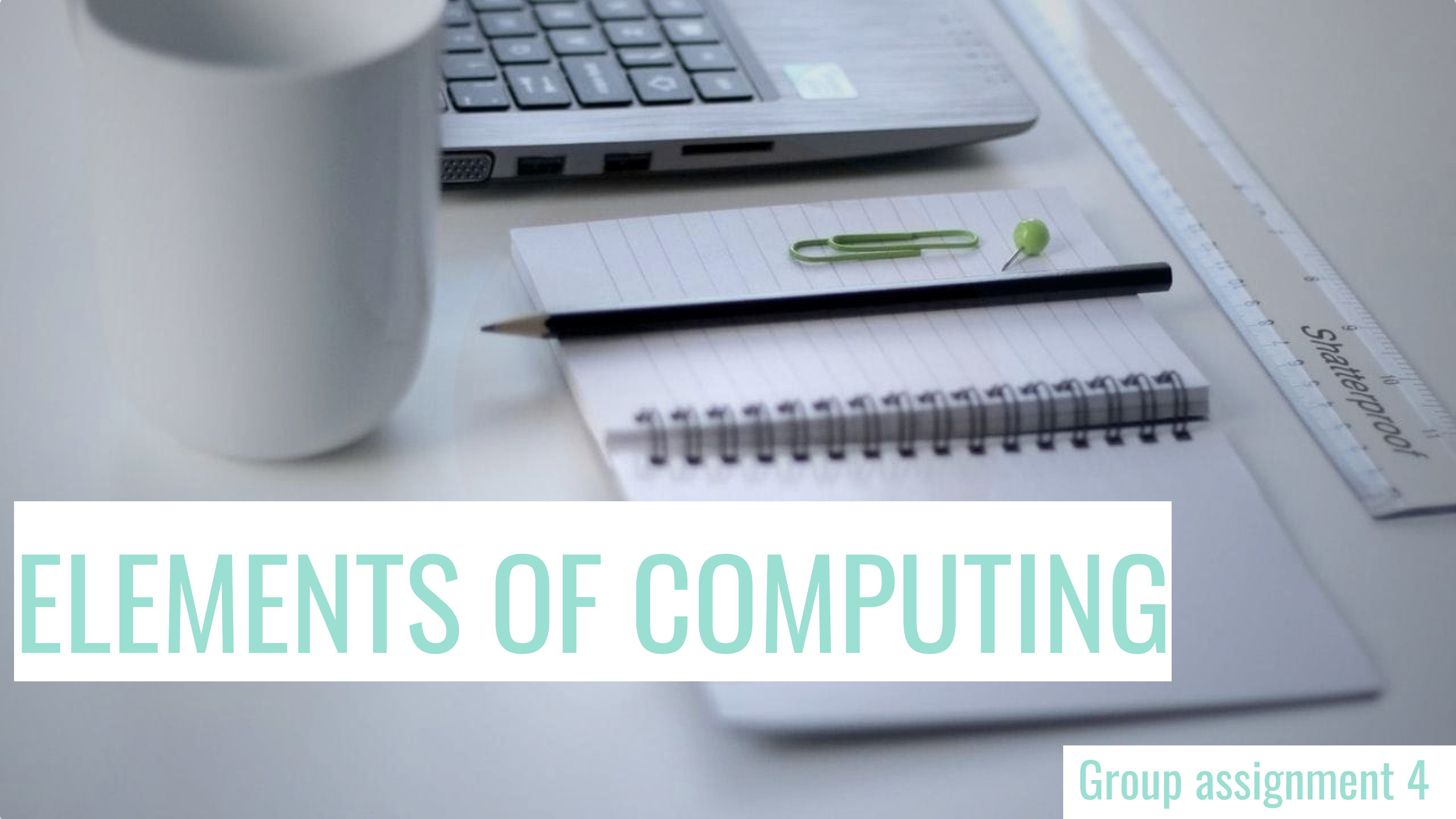


ELEMENTS OF COMPUTING

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)

1

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

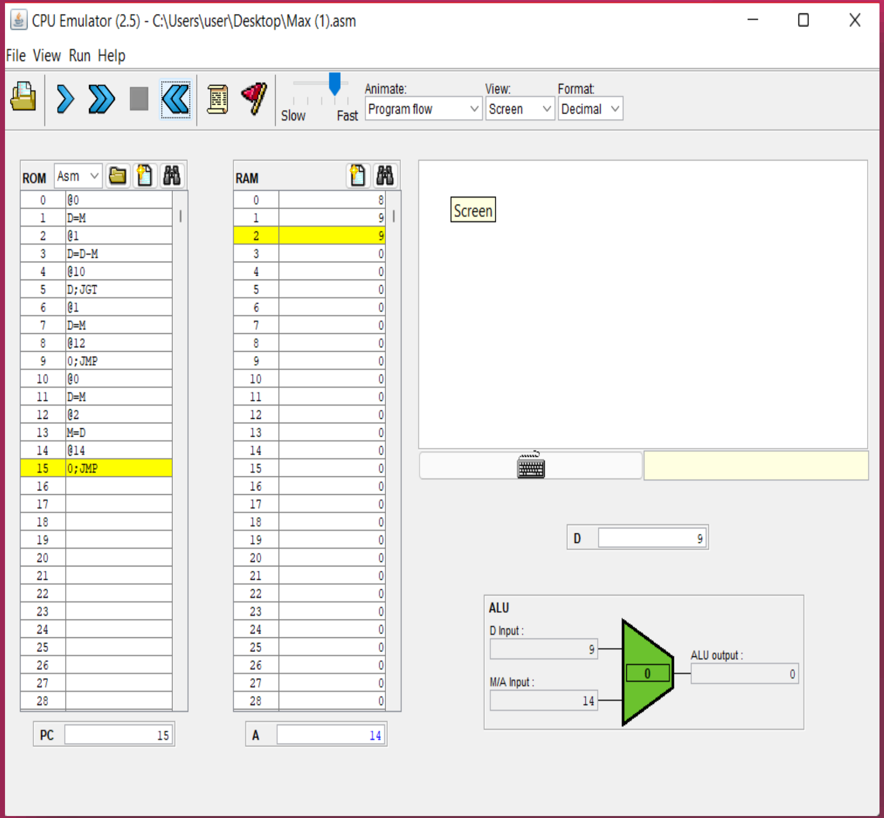
```
// Symbol-less version of the Max.asm program.

 @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
```

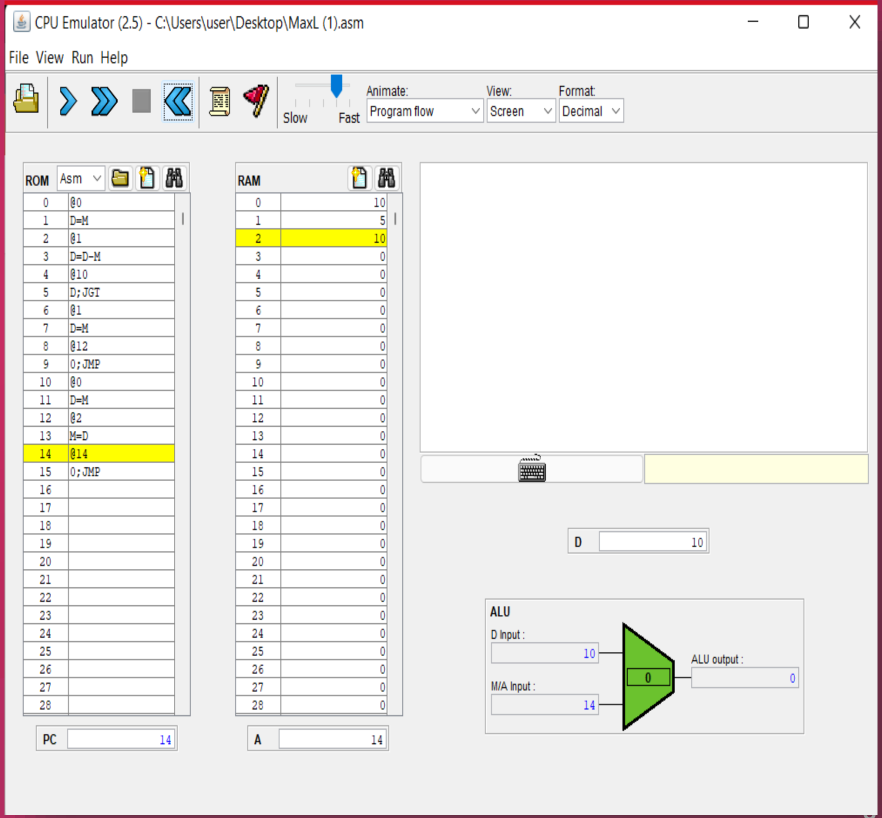
Output of MAX.asm file

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



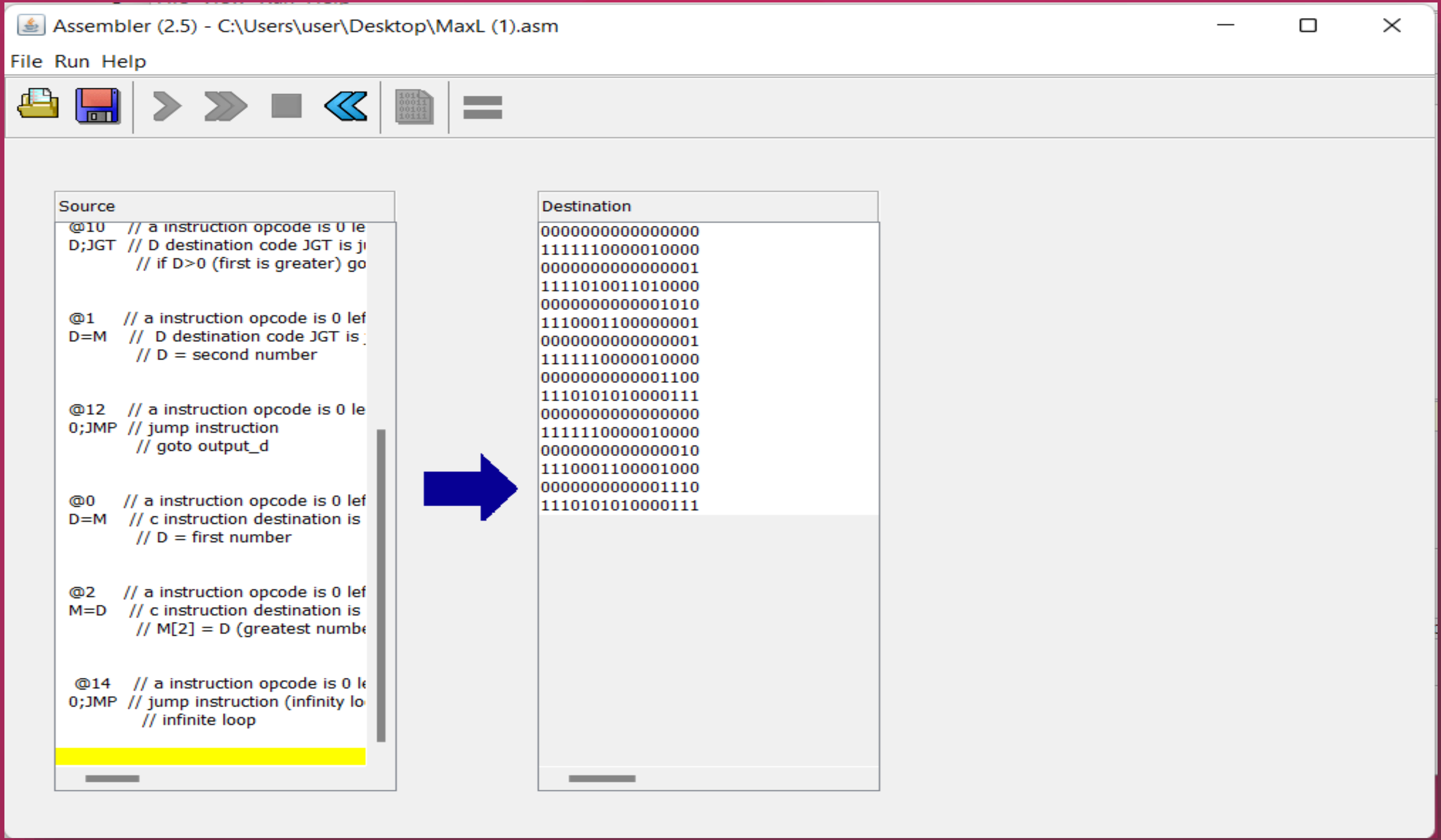
Output of MAXL.asm file

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



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 ,comparision and jump as comments.

```
0000000000000000    // A-instruction: @0= @valueinbinary= 0000000000000000
1111110000010000    // C-instruction: dest=D;comp=M;jump=null: 1111110000010000
0000000000000001    // A-instruction: @1= @valueinbinary= 0000000000000001
1111010011010000    // C-instruction: dest=D;comp=D-M;jump=null: 1111010011010000
0000000000001010    // A-instruction: @10= @valueinbinary= 0000000000001010
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: 1111110000010000
0000000000001100    // A-instruction: @12= @valueinbinary= 0000000000001100
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: 1111110000010000
0000000000000010    // A-instruction: @2= @valueinbinary= 0000000000000010
1110001100001000    // C-instruction: dest=M;comp=D;jump=null: 1110001100001000
0000000000001110    // A-instruction: @14= @valueinbinary= 0000000000001110
1110101010000111    // C-instruction: dest=null;comp=0;jump=JMP: 1110101010000111
```

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

<u>Python Code:</u>

EXPLORER

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

Assembler.py

EOC_Assignment > Assembler.py > ...

```python
import sys
import Code
import Parser
import SymbolTable


class Assembler:
    """
    Reads Progam.asm source file and creates a new file Program.hack which has the assembled machine code as a text file.
    The Assembly is implemented as two stages or two passes. The first pass scans the whole program and registers
    symbols in the symbol table. The second pass scans the whole program again substituting the symbols with their
    respective addresses in the symbol table, in addition to generating binary machine code and writing the resulting
    assembled machine code to a new file.
    Usage: python Assembler.py Program.asm
    """
    def __init__(self):
        self.symbol_address = 16
        self.symbols_table = SymbolTable.SymbolTable()

    @staticmethod
    def get_hack_file(asm_file):
        """
        Suggests a file name for the Hack Machine Code source file.
        :param asm_file: Program source code file written in Hack Assembly Language.
        :return: String.
        """
        if asm_file.endswith('.asm'):
            return asm_file.replace('.asm', '.hack')
        else:
            return asm_file + '.hack'
```

EXPLORER

PYTHON

- EOC_Assignment
  - Assembler.py
  - Code.py
  - Lex.py
  - Max.asm
  - MaxL.asm
  - Parser.py
  - SymbolTable.py

Assembler.py

EOC_Assignment > Assembler.py > ...

```python

    def _get_address(self, symbol):
        """
        Helper method. Looks-up the address of a symbol (decimal value, label or variable).
        :param symbol: Symbol or Value.
        :return: Address.
        """
        if symbol.isdigit():
            return symbol
        else:
            if not self.symbols_table.contains(symbol):
                self.symbols_table.add_entry(symbol, self.symbol_address)
                self.symbol_address += 1
            return self.symbols_table.get_address(symbol)

    def pass_1(self, file):
        """
        First compilation pass: Determine memory locations of label definitions: (LABEL).
        :param file:
        :return:
        """
        parser = Parser.Parser(file)
        curr_address = 0
        while parser.has_more_instructions():
            parser.advance()
            inst_type = parser.instruction_type
            if inst_type in [parser.A_INSTRUCTION, parser.C_INSTRUCTION]:
                curr_address += 1
            elif inst_type == parser.L_INSTRUCTION:
                self.symbols_table.add_entry(parser.symbol, curr_address)
```

EXPLORER

PYTHON

EOC_Assignment
- Assembler.py
- Code.py
- Lex.py
- Max.asm
- MaxL.asm
- Parser.py
- SymbolTable.py

OUTLINE

TIMELINE

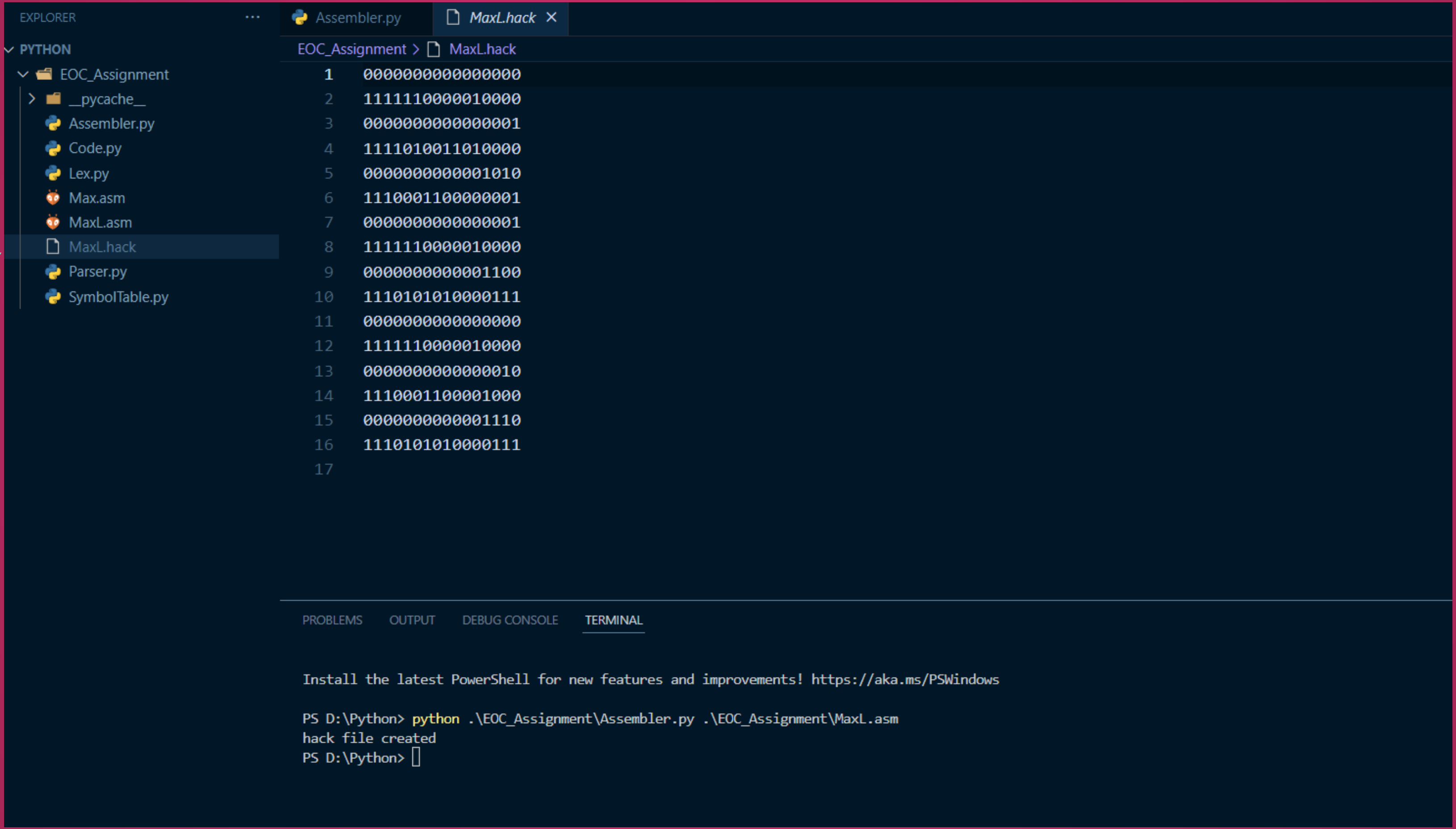
JAVA PROJECTS

Assembler.py

EOC_Assignment > Assembler.py > ...

```python
    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 Asembly 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.jmp) + '\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')
```

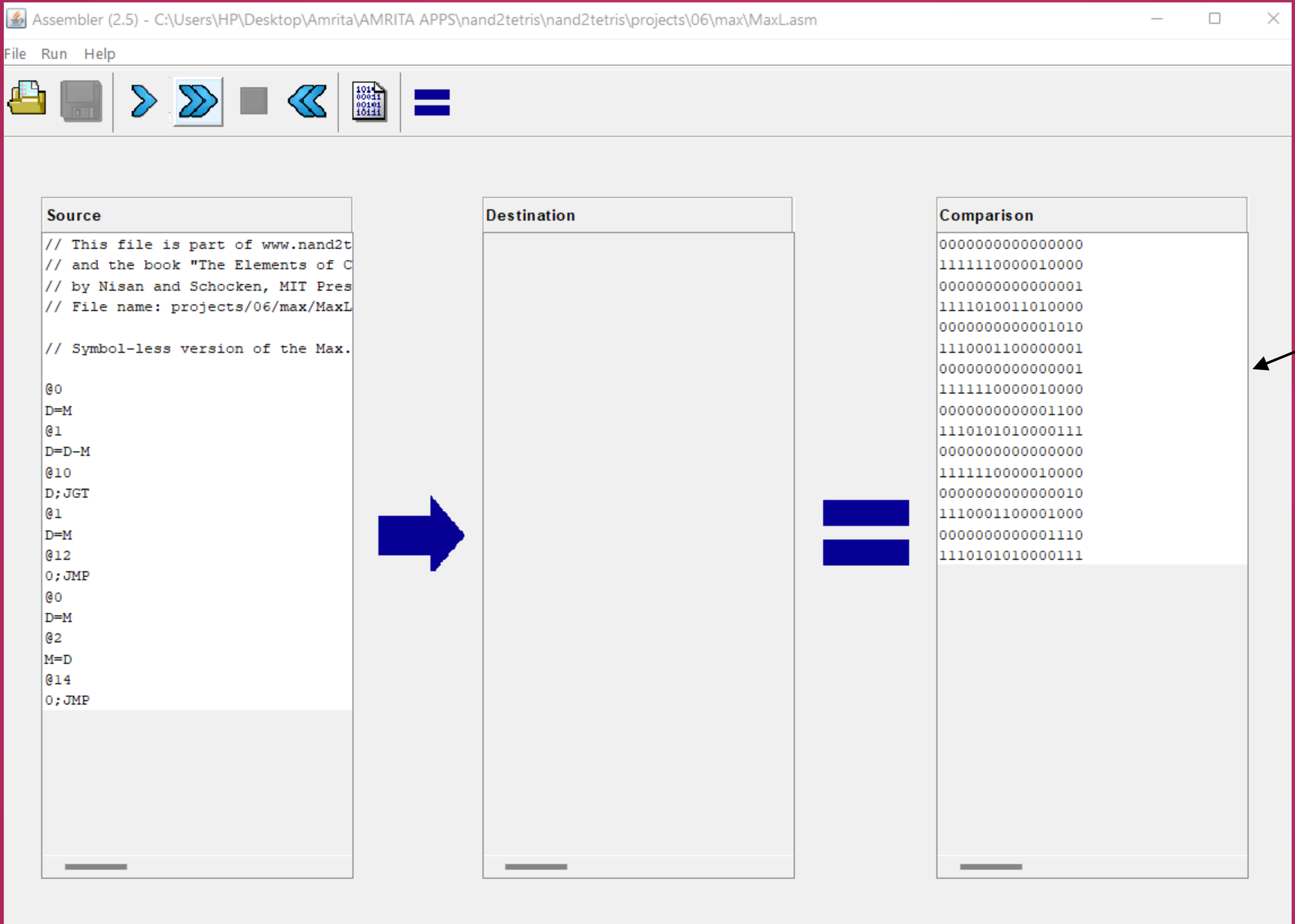
Output: Output obtained when an .asm file is passed through the code



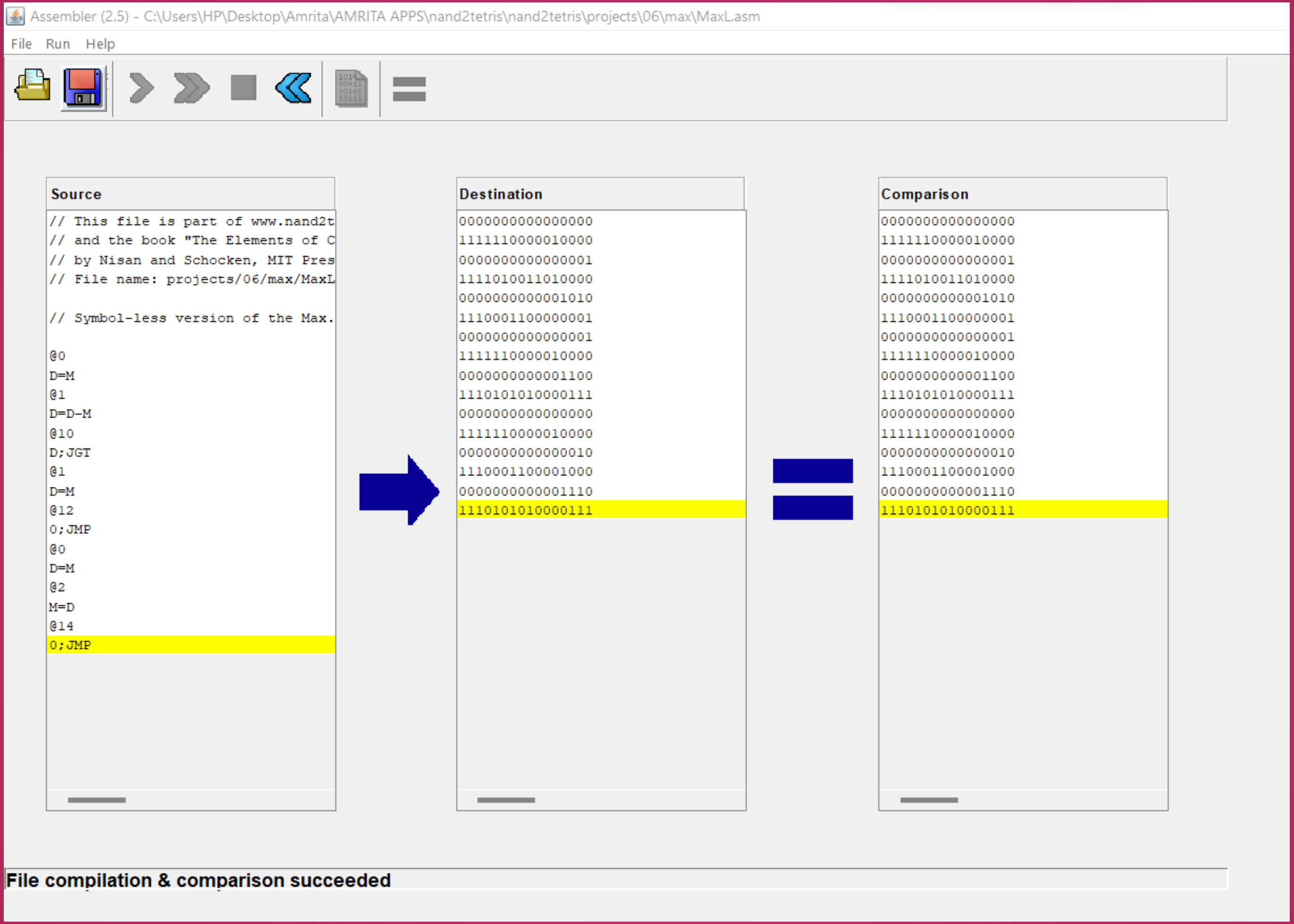
Hack file appears

.hack file for MaxL.asm is created

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



File created using python code



Note: All the essential python files required to create an assembler are included in submission folder.

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

Python Code:

EXPLORER ...

PYTHON

EOC_Assignment
- Assembler.py
- Code.py
- Lex.py
- Max.asm
- MaxL.asm
- Parser.py
- SymbolTable.py

Assembler.py ×

EOC_Assignment > Assembler.py > ...

```python
import sys
import Code
import Parser
import SymbolTable


class Assembler:
    """
    Reads Progam.asm source file and creates a new file Program.hack which has the assembled machine code as a text file.
    The Assembly is implemented as two stages or two passes. The first pass scans the whole program and registers
    symbols in the symbol table. The second pass scans the whole program again substituting the symbols with their
    respective addresses in the symbol table, in addition to generating binary machine code and writing the resulting
    assembled machine code to a new file.
    Usage: python Assembler.py Program.asm
    """
    def __init__(self):
        self.symbol_address = 16
        self.symbols_table = SymbolTable.SymbolTable()

    @staticmethod
    def get_hack_file(asm_file):
        """
        Suggests a file name for the Hack Machine Code source file.
        :param asm_file: Program source code file written in Hack Assembly Language.
        :return: String.
        """
        if asm_file.endswith('.asm'):
            return asm_file.replace('.asm', '.hack')
        else:
            return asm_file + '.hack'
```

EXPLORER ···

PYTHON

- EOC_Assignment
  - Assembler.py
  - Code.py
  - Lex.py
  - Max.asm
  - MaxL.asm
  - Parser.py
  - SymbolTable.py

Assembler.py ×

EOC_Assignment > Assembler.py > ...

```python
    def _get_address(self, symbol):
        """
        Helper method. Looks-up the address of a symbol (decimal value, label or variable).
        :param symbol: Symbol or Value.
        :return: Address.
        """
        if symbol.isdigit():
            return symbol
        else:
            if not self.symbols_table.contains(symbol):
                self.symbols_table.add_entry(symbol, self.symbol_address)
                self.symbol_address += 1
            return self.symbols_table.get_address(symbol)

    def pass_1(self, file):
        """
        First compilation pass: Determine memory locations of label definitions: (LABEL).
        :param file:
        :return:
        """
        parser = Parser.Parser(file)
        curr_address = 0
        while parser.has_more_instructions():
            parser.advance()
            inst_type = parser.instruction_type
            if inst_type in [parser.A_INSTRUCTION, parser.C_INSTRUCTION]:
                curr_address += 1
            elif inst_type == parser.L_INSTRUCTION:
                self.symbols_table.add_entry(parser.symbol, curr_address)
```

EXPLORER

PYTHON

- EOC_Assignment
  - Assembler.py
  - Code.py
  - Lex.py
  - Max.asm
  - MaxL.asm
  - Parser.py
  - SymbolTable.py

OUTLINE

TIMELINE

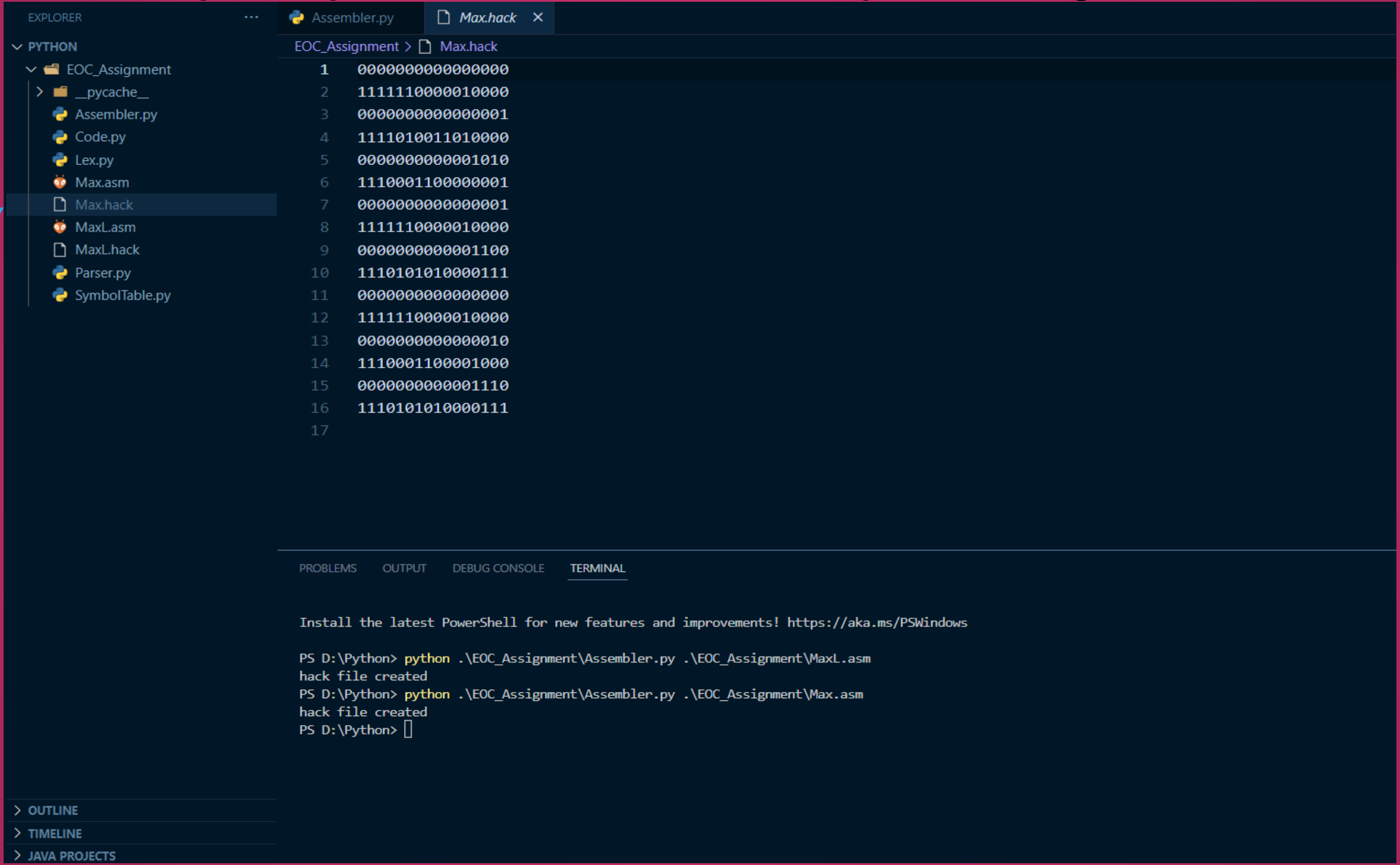
JAVA PROJECTS

Assembler.py

EOC_Assignment > Assembler.py > ...

```python
    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 Asembly 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.jmp) + '\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')
```

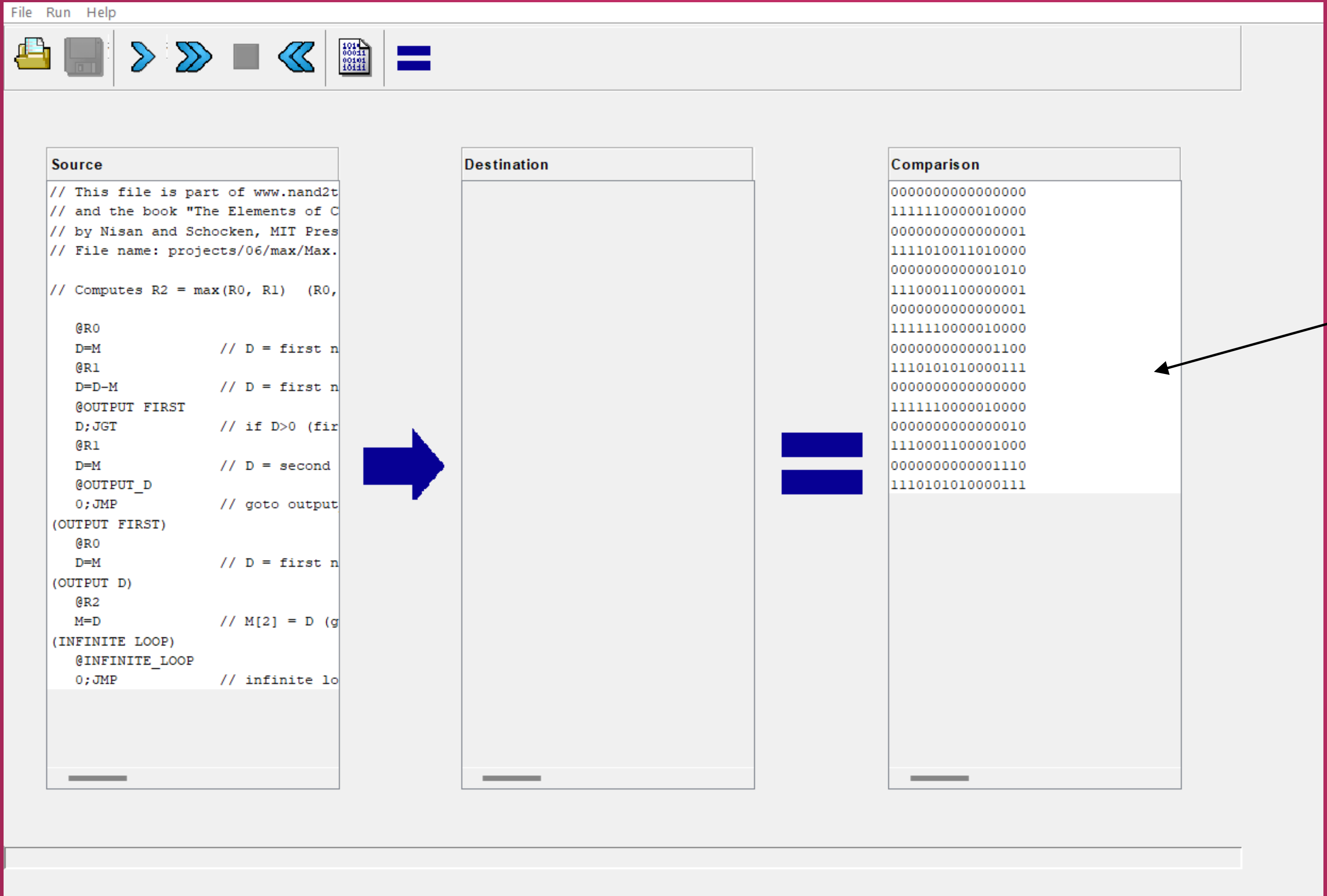
Output: Output obtained when an .asm file is passed through the code



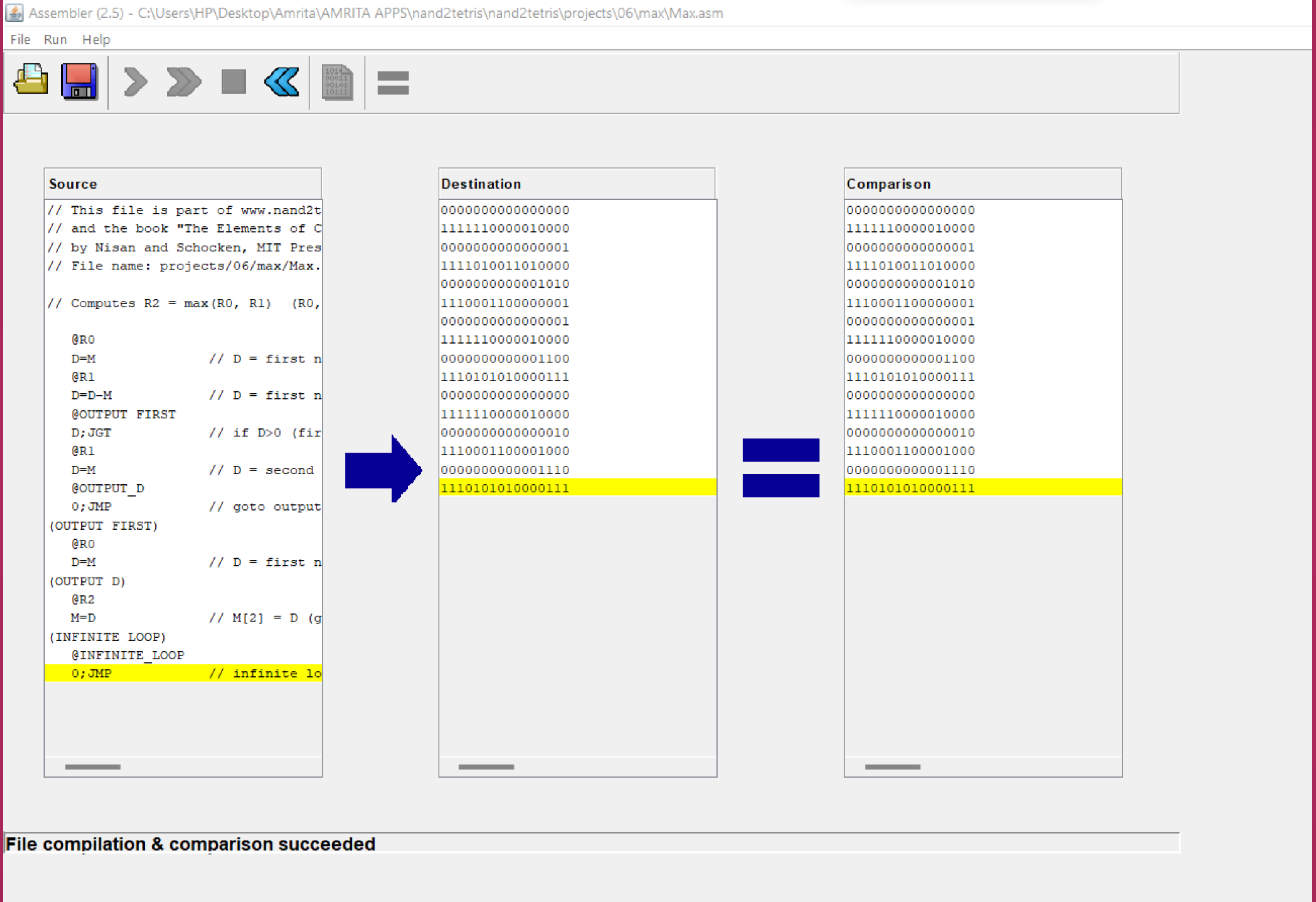
Hack file appears

.hack file for Max.asm is created

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



File created using python code


Assembler (2.5) - C:\Users\HP\Desktop\Amrita\AMRITA APPS\nand2tetris\nand2tetris\projects\06\max\Max.asm
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/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
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

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
```

EXPLAINATION

- *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 @I 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 @I as we progress in the loop, I value increments by 1 and by this loop and the code we use we will find the sum .*
- *The loop continues until I value is greater than n , n will be taken as our wish.*

CONTINUTION

- 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
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
```

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

```
0000000000010000
1111010011010000
0000000000011000
1110001100000001
0000000000010010
1111110000010000
0000000000010001
1111000010010000
0000000000010010
1110001100001000
0000000000010001
1111110111001000
0000000000001000
1110101010000111
0000000000010010
1111110000010000
0000000000010011
1110101010001000
0000000000000000
1111010011010000
0000000000100110
1110001100000100
0000000000010011
1111110111001000
0000000000010100
1110001100001000
0000000000011100
1110001100000001
0000000000100110
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)
0000000000010000 // it is a instruction so binary representation of 16 is represented
1110001100001000 // a = 0 ; c = 001100 ; d = 001 ; j = 000
0000000000010001 // it is a instruction so binary representation of 17 is represented
1110101010001000 // a = 0 ; c = 101010 ; d = 001 ; j = 000
0000000000010010 // it is a instruction so binary representation of 18 is represented
1110101010001000 // a = 0 ; c = 101010 ; d = 001 ; j = 000
0000000000010001 // it is a instruction so binary representation of 17 is represented
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000
0000000000010000 // it is a instruction so binary representation of 16 is represented
1111010011010000 // a = 1; c = 010011 ; d = 010 ; j = 000
0000000000011000 // it is a instruction so binary representation of 24 is represented
1110001100000001 // a = 0 ; c = 001100 ;d = 000 ; j = 001
0000000000010010 // it is a instruction so binary representation of 18 is represented
1111110000010000 // a = 1 ; c = 110000 ; d = 010 ; j = 000
0000000000010001 // it is a instruction so binary representation of 17 is represented
1111000010010000 // a = 1 ; c = 000010 ; d = 010 ; j = 000
0000000000010010 // it is a instruction so binary representation of 18 is represented
1110001100001000 // a = 0 ; c = 001100 ; d = 001 ; j = 000
0000000000010001 // it is a instruction so binary representation of 17 is represented
1111110111001000 // a = 1 ; c = 110111 ; d = 001 ; j= 000
0000000000001000 // it is a instruction so binary representation of 8 is represented
1110101010000111 // a = 0 ; c = 101010 ; d = 000 ; j = 111
0000000000010010 // it is a instruction so binary representation of 18 is represented
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
```

```
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
0000000000100110 // 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
0000000000100110 // 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

```
0000000000010000
1111010011010000
0000000000011000
1110001100000001
0000000000010010
1111110000010000
0000000000010001
1111000010010000
0000000000010010
1110001100001000
0000000000010001
1111110111001000
0000000000001000
1110101010000111
0000000000010010
1111110000010000
0000000000010011
1110101010001000
0000000000000000
1111010011010000
0000000000100110
1110001100000100
0000000000010011
1111110111001000
0000000000010100
1110001100001000
0000000000011100
1110001100000001
0000000000100110
1110101010000111
```

Comparison

```
0000000000010000
1111010011010000
0000000000011000
1110001100000001
0000000000010010
1111110000010000
0000000000010001
1111000010010000
0000000000010010
1110001100001000
0000000000010001
1111110111001000
0000000000001000
1110101010000111
0000000000010010
1111110000010000
0000000000010011
1110101010001000
0000000000000000
1111010011010000
0000000000100110
1110001100000100
0000000000010011
1111110111001000
0000000000010100
1110001100001000
0000000000011100
1110001100000001
0000000000100110
1110101010000111
```

ile compilation & comparison succeeded

THANK YOU