IN0013 Problem solving assessment
Working Answers

Name: Rodin Opuz
Student ID: 240016064

Question 1

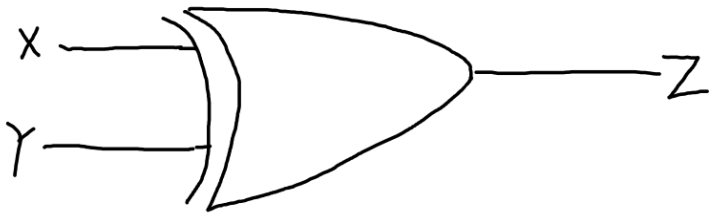
1a)

A Truth table in the context of Logic gates displays the inputs and outputs of Boolean expressions. A Truth table contains all possible inputs and outputs for a Boolean expression. Each input and output represents either 0 or 1.

1b)

Working space (A.B) + (B.NOT C)						
A	B	C	NOT C	A AND B	NOT C AND B	X
0	0	0	1	0	0	0
0	0	1	0	0	0	0
0	1	0	1	0	1	1
0	1	1	0	0	0	0
1	0	0	1	0	0	0
1	0	1	0	0	0	0
1	1	0	1	1	1	1
1	1	1	0	1	0	1

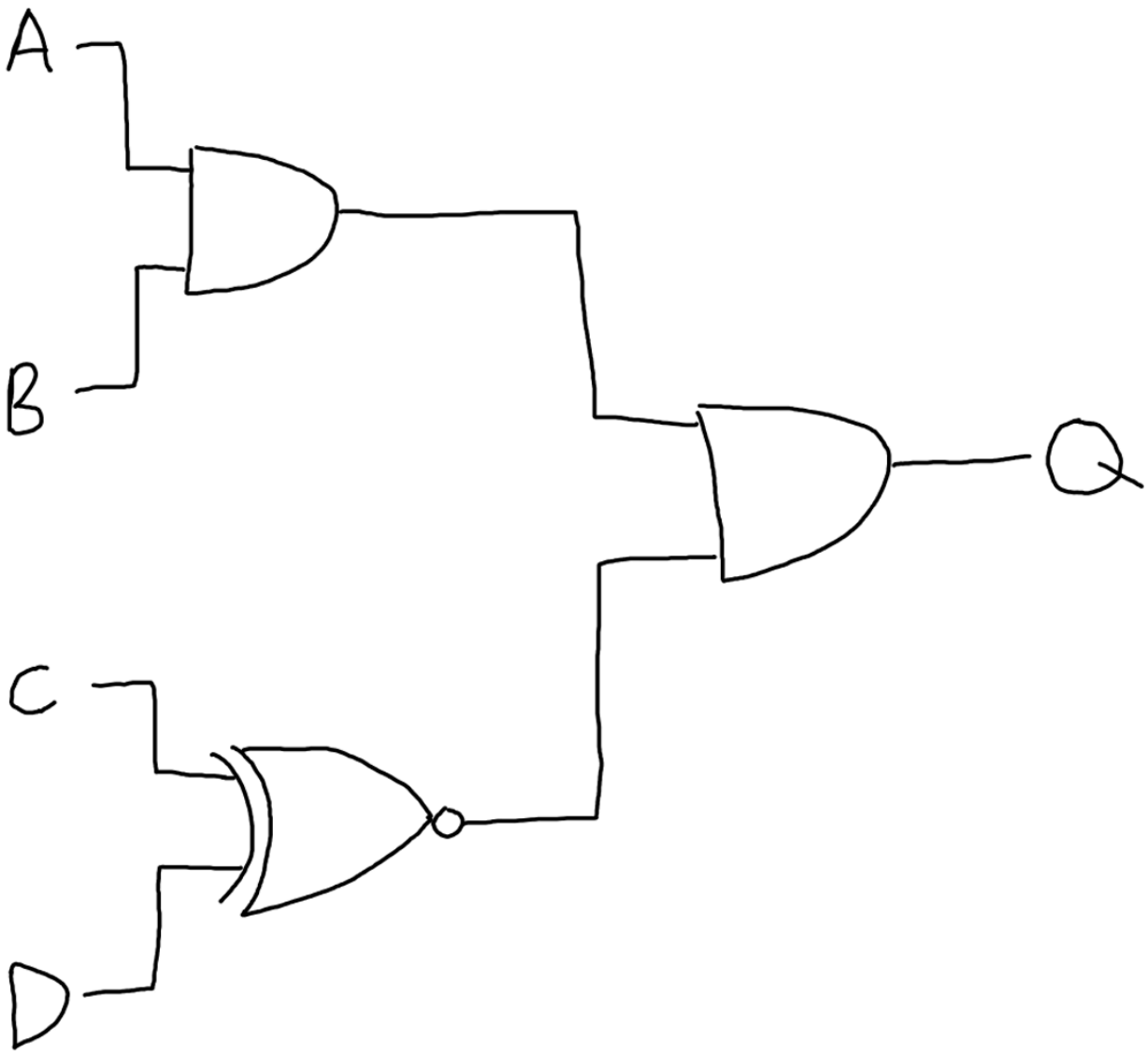
1c)



An XOR gate only outputs True if only one of the two inputs are True. XOR means exclusively OR and the function is displayed with the truth table below.

X	Y	Z
0	0	0
0	1	1
1	0	1
1	1	0

Question 2



Q = (A AND B) AND (C XNOR D)

There are two main conditions for Q to be HIGH.

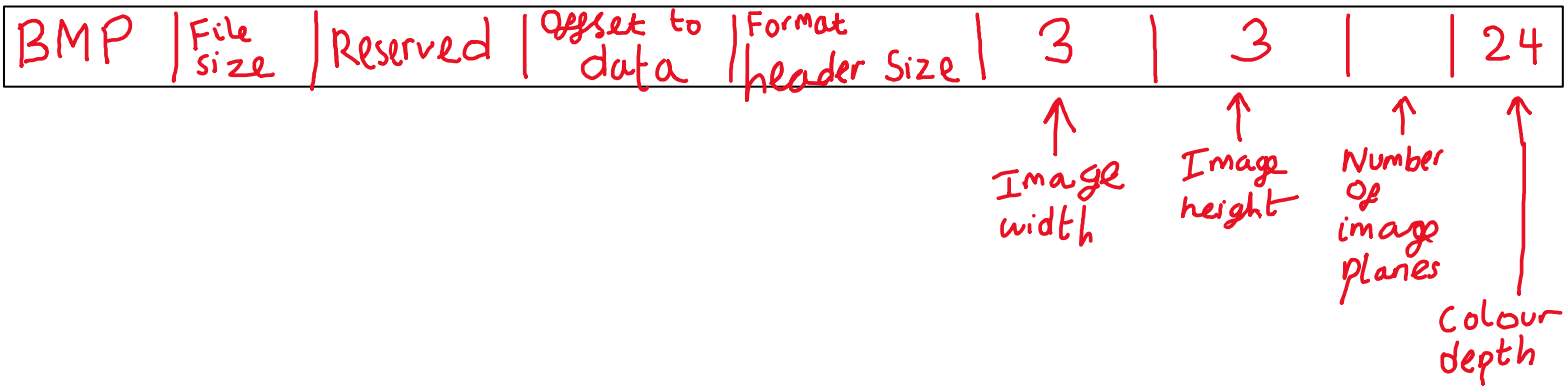
The first condition is when A AND B are both high. The second condition is when either C and D are both high or both low. The first condition can be represented as Q1 = (A AND B). The second condition can be represented as Q2 = (C XNOR D). Then for Q to be HIGH we need both conditions to both be HIGH so Q = Q1 AND Q2.

$$Q = (A \cdot B) \cdot \overline{(C \oplus D)}$$

Question 3

3a)

A BMP file (Bitmap file) of a 3x3 image and 24 bits colour depth contains metadata. The metadata is the total file size, reserved, offset to data, format header size, image width, image height, number of image planes and colour depth.



3b)

The first bit represents red, the second bit represents blue, the third bit represents green (RGB). Yellow = {110}, white = {111} and black = {000}. 0 represents RGB value 0 and 1 represents RGB value 255. The metadata includes the image dimensions (3 x 3 image) and colour depth (3 bits per pixel).

IMAGE = {Black, Yellow, Black}, {White, Yellow, White}, {Yellow, Black, Yellow}
IMAGE = {000, 110, 000}, {111, 110, 111}, {110, 000, 110}
Flat-Level Bit-String = 000110000111110111110000110
This is the metadata of the colours for the image.

Question 4

```python
# This creates the class Employee.
class Employee:
    def __init__(self, emp_name, emp_id, emp_salary, emp_department): # This creates the attributes, emp_name, emp_id, em
        self.emp_name = emp_name # This assigns the attribute emp_name to the variable. The name of the employee.
        self.emp_id = emp_id # This assigns the attribute emp_id to the variable. The ID of the employee.
        self.emp_salary = emp_salary # This assigns the attribute emp_salary to the variable. The salary of the employee.
        self.emp_department = emp_department # This assigns the attribute emp_department to the variable. The department

    # This is the method to calculate the salary and calculates overtime.
    def calculate_emp_salary(self, salary, hours_worked): # Creates the attributes salary and hours_worked.
        # If the hours worked by the employee is bigger than 50, it calculates overtime pay.
        if hours_worked > 50:
            overtime = hours_worked - 50 # This calculates how many overtime hours the employee has worked.
            overtime_amount = overtime * (salary / 50) # This calculates the pay based on the amount of overtime hours.
        salary_change = salary + overtime_amount # Adds the overtime pay to the salary.
        self.emp_salary = salary_change # This stores the salary change in the employee salary attribute.
        return salary_change # Outputs the salary.

    # This is the method to change an employee's department if needed.
    def emp_assign_department(self, department_change): # Creates the department_change attribute.
        self.emp_department = department_change # This stores the department change in the employee department attribute.

    # This is the method to display employee details
    def print_employee_details(self):
        print(f"Name:    {self.emp_name}") # Prints the name of the employee.
        print(f"ID:    {self.emp_id}") # Prints the ID of the employee.
        print(f"Salary:    {self.emp_salary}") # Prints the salary of the employee.
        print(f"Department:    {self.emp_department}") # Prints the department of the employee.
        print("------------------------") # To make it similar to the output example in the worksheet.
        print() # Spaces out each employee to make it more readable.


#TESTING

emp_1 = Employee(emp_name = "ADAMS", emp_id = "E7876", emp_salary = 50000, emp_department = "ACCOUNTING")
emp_2 = Employee(emp_name = "JONES", emp_id = "E7499", emp_salary = 45000, emp_department = "RESEARCH")

emp_1.print_employee_details()
emp_2.print_employee_details()

emp_1.emp_assign_department("CEO")
emp_2.emp_assign_department("ACCOUNTING")
emp_1_salary_change = emp_1.calculate_emp_salary(emp_1.emp_salary, 2000)

emp_1.print_employee_details()
emp_2.print_employee_details()
```

```
Python 3.13.0 (v3.13.0:60
Type "help", "copyright",
>>>
============== RESTART:
>>>
============== RESTART:
Name:    ADAMS
ID:    E7876
Salary:    50000
Department:    ACCOUNTING
------------------------

Name:    JONES
ID:    E7499
Salary:    45000
Department:    RESEARCH
------------------------

Name:    ADAMS
ID:    E7876
Salary:    2000000.0
Department:    CEO
------------------------

Name:    JONES
ID:    E7499
Salary:    45000
Department:    ACCOUNTING
------------------------
>>>
```

This has the Python class Employee with the attributes emp_id, emp_name, emp_salary, and emp_department. It has 3 methods calculate_emp_salary, emp_assign_department and print_employee_details. Each method does a different thing. The method calculate_emp_salary is used to calculate an employee's salary based on additional overtime pay. If an employee works more than 50 hours, they will get overtime pay added to their overall salary. The emp_assign_department is used to update an employee's department. The method print_employee_details is used to output the name of the employee, the id of the employee, the salary of the employee, and the department of the employee.

```
# This creates the class Employee.
class Employee:
    def __init__(self, emp_name, emp_id, emp_salary, emp_department): # This creates the attributes, emp_name, emp_id, emp_salary and emp_department.
        self.emp_name = emp_name # This assigns the attribute emp_name to the variable. The name of the employee.
        self.emp_id = emp_id # This assigns the attribute emp_id to the variable. The ID of the employee.
        self.emp_salary = emp_salary # This assigns the attribute emp_salary to the variable. The salary of the employee.
        self.emp_department = emp_department # This assigns the attribute emp_department to the variable. The department the employee is in.

    # This is the method to calculate the salary and calculates overtime.
    def calculate_emp_salary(self, salary, hours_worked): # Creates the attributes salary and hours_worked.
        # If the hours worked by the employee is bigger than 50, it calculates overtime pay.
        if hours_worked > 50:
            overtime = hours_worked - 50 # This calculates how many overtime hours the employee has worked.
            overtime_amount = overtime * (salary / 50) # This calculates the pay based on the amount of overtime hours.
        salary_change = salary + overtime_amount # Adds the overtime pay to the salary.
        self.emp_salary = salary_change # This stores the salary change in the employee salary attribute.
        return salary_change # Outputs the salary.

    # This is the method to change an employee's department if needed.
    def emp_assign_department(self, department_change): # Creates the department_change attribute.
        self.emp_department = department_change # This stores the department change in the employee department attribute.
    # This is the method to display employee details
    def print_employee_details(self):
        print(f"Name:    {self.emp_name}") # Prints the name of the employee.
        print(f"ID:    {self.emp_id}") # Prints the ID of the employee.
        print(f"Salary:    {self.emp_salary}") # Prints the salary of the employee.
        print(f"Department:    {self.emp_department}") # Prints the department of the employee.
        print("------------------------") # To make it similar to the output example in the worksheet.
        print() # Spaces out each employee to make it more readable.

'''
#TESTING
emp_1 = Employee(emp_name = "ADAMS", emp_id = "E7876", emp_salary = 50000, emp_department = "ACCOUNTING")
emp_2 = Employee(emp_name = "JONES", emp_id = "E7499", emp_salary = 45000, emp_department = "RESEARCH")
emp_1.print_employee_details()
emp_2.print_employee_details()

emp_1.emp_assign_department("CEO")
emp_2.emp_assign_department("ACCOUNTING")
emp_1_salary_change = emp_1.calculate_emp_salary(emp_1.emp_salary, 2000)

emp_1.print_employee_details()
emp_2.print_employee_details()
'''
```

Question 5

The code creates two classes, Polygon and Triangle. The Triangle class inherits from Polygon (Single inheritance - 1 to 1). Polygon is a parent class and Triangle is a child class.

1 Polygon creates a method with the arguments, self and no_of_sides. Self represents the reference to the current instance of the class and no_of_sides represents the amount of sides the polygon has. The method creates a list of side lengths based on the number of sides (no_of_sides) and each one is 0 temporarily. The method inputSides asks the user to input the length of every side in the polygon and it stores each input in the list. The method dispSides outputs the length of each side based on the inputs of the user.

2 Triangle inherits all the methods and attributes of the Polygon class. The method calls Polygon, and the no_of_sides argument equals 3 which represents how many sides a triangle has. The method findPerimeter has a, b and c and those variables store each side of the triangle. Three variables for three side lengths of the triangle. The perimeter variable adds the three variables, a, b and c which hold the side lengths together. This outputs the print statement "The perimeter of the triangle is "and the perimeter of the triangle.

The program uses encapsulation, abstraction, inheritance and polymorphism. The Triangle class inherits from the Polygon. The class Triangle uses the methods and attributes of Polygon and also use methods to calculate perimeter which is an example of polymorphism. The program uses abstraction to hide the implementation and only show the input and output.

1
```
class Polygon:
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is", self.sides[i])
```

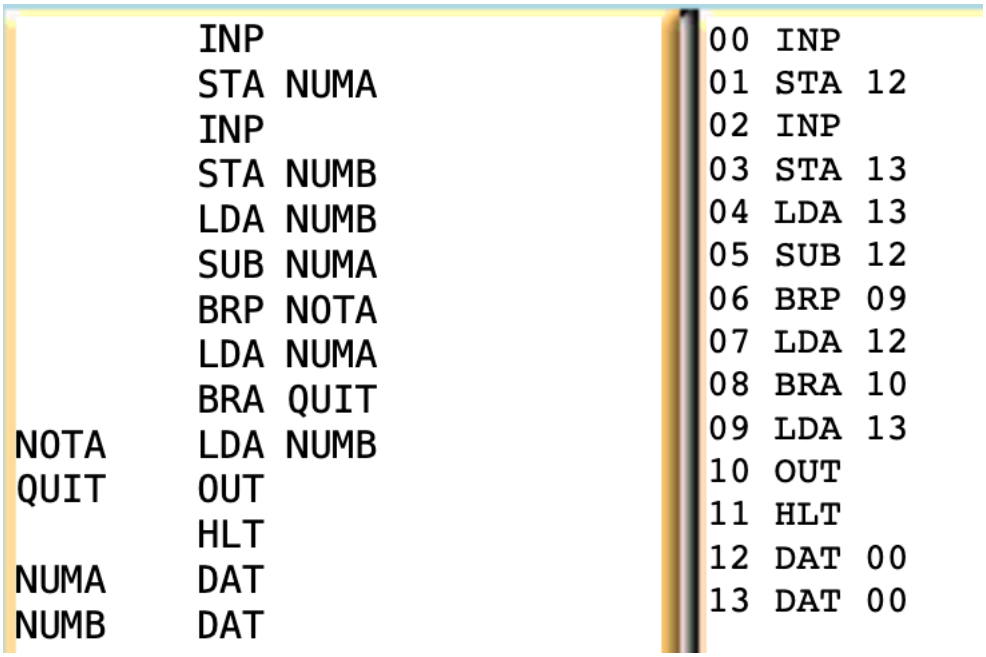
2
```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findPerimeter(self):
        a, b, c = self.sides
        Perimeter = a+b+c
        print('The perimeter of the triangle is %0.2f' %Perimeter)
```

```
triangle_1 = Triangle()
triangle_1.inputSides()
triangle_1.dispSides()
triangle_1.findPerimeter()
```

} Additional code

Question 6

6a)



```
          INP
          STA NUMA
          INP
          STA NUMB
          LDA NUMB
          SUB NUMA
          BRP NOTA
          LDA NUMA
          BRA QUIT
NOTA      LDA NUMB
QUIT      OUT
          HLT
NUMA      DAT
NUMB      DAT
```

```
00 INP
01 STA 12
02 INP
03 STA 13
04 LDA 13
05 SUB 12
06 BRP 09
07 LDA 12
08 BRA 10
09 LDA 13
10 OUT
11 HLT
12 DAT 00
13 DAT 00
```

- I added LDA NUMB to load the value from NUMB.
- I changed LDA NUMB to LDA NUMA to load the value from NUMA.
- I changed NOTA LDA NUMA to NOTA LDA NUMB so if the result of NUMB – NUMA is positive or zero, it loads the value from NUMB. The value from NUMB is loaded if the second number is greater than the first number.

Pseudocode program

```
NUMA = INPUT()
NUMB = INPUT()
IF NUMB – NUMA >= 0
        OUTPUT NUMB
ELSE
        OUTPUT NUMA
```

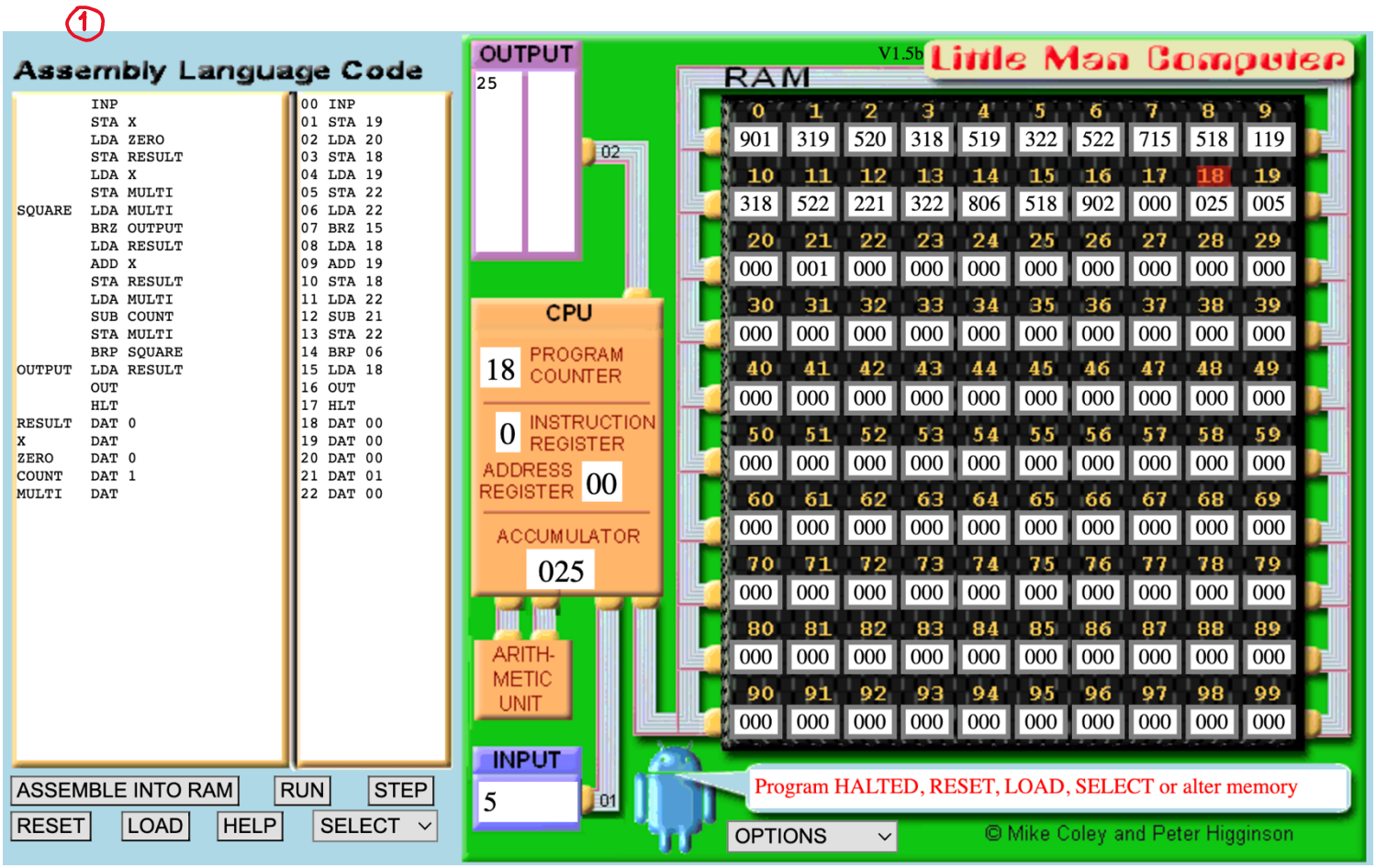
Example

NUMA = 7

NUMB = 12

12 – 7 >= 0 so output NUMB. It outputs 12.

6b)



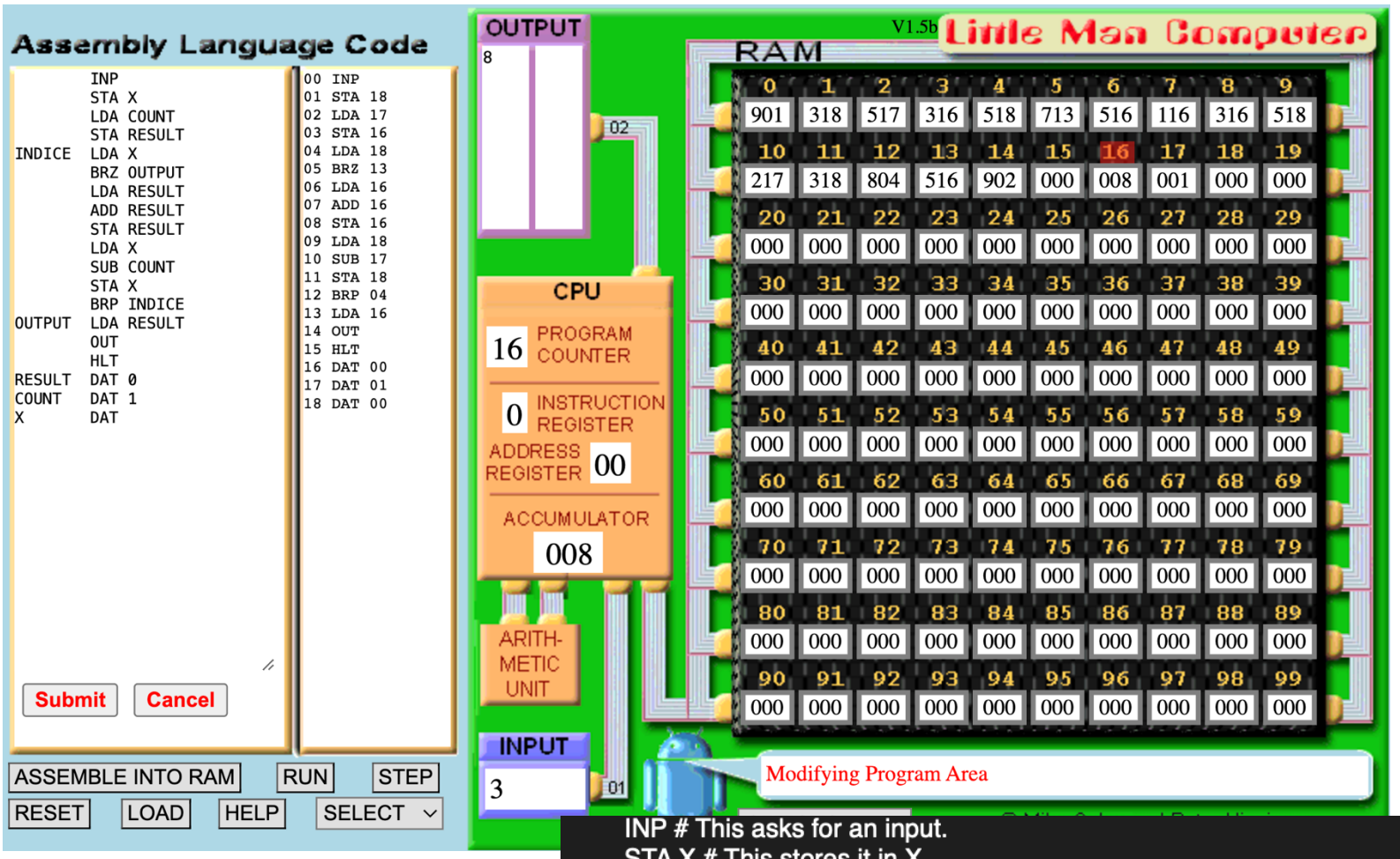
X^2

This takes a number as input and stores it in X. It stores 0 in RESULT AND stores the input from X in MULTI. It adds X to the RESULT an X number of times. Every time X is added to RESULT, it subtracts 1 from MULTI. This is because in LMC, to square or multiply a number it uses repeating addition. Once MULTI reaches 0, the repeating addition is stopped. It outputs once the addition is done.

5 ^ 2 = 25

```
      INP # This asks for an input from the user.
      STA X # This stores it in X.
      LDA ZERO # This loads zero which contains the value 0.
      STA RESULT # This stores zero in RESULT.
      LDA X # This loads X.
      STA MULTI # This stores the value in X in MULTI.
SQUARE LDA MULTI # This loads MULTI.
      BRZ OUTPUT # If MULTI is 0, it jumps to OUTPUT.
      LDA RESULT # This loads the current value of RESULT.
      ADD X # This adds X to the RESULT.
      STA RESULT # This stores it in RESULT.
      LDA MULTI # This loads MULTI.
      SUB COUNT # This subtracts 1 from MULTI.
      STA MULTI # This stores the value after it was subtracted in MULTI.
      BRP SQUARE # If MULTI is positive, it will jump to SQUARE
OUTPUT  LDA RESULT # This loads RESULT.
      OUT # This outputs the value in RESULT.
      HLT # This halts the program.

# This stores 0 in RESULT at the start.
# This creates space for X.
# This stores 0 in ZERO at the start.
 # This stores 1 in COUNT at the start.
# This creates space for MULTI.
RESULT  DAT 0
X       DAT
ZERO    DAT 0
COUNT   DAT 1
MULTI   DAT
```

2 2^X



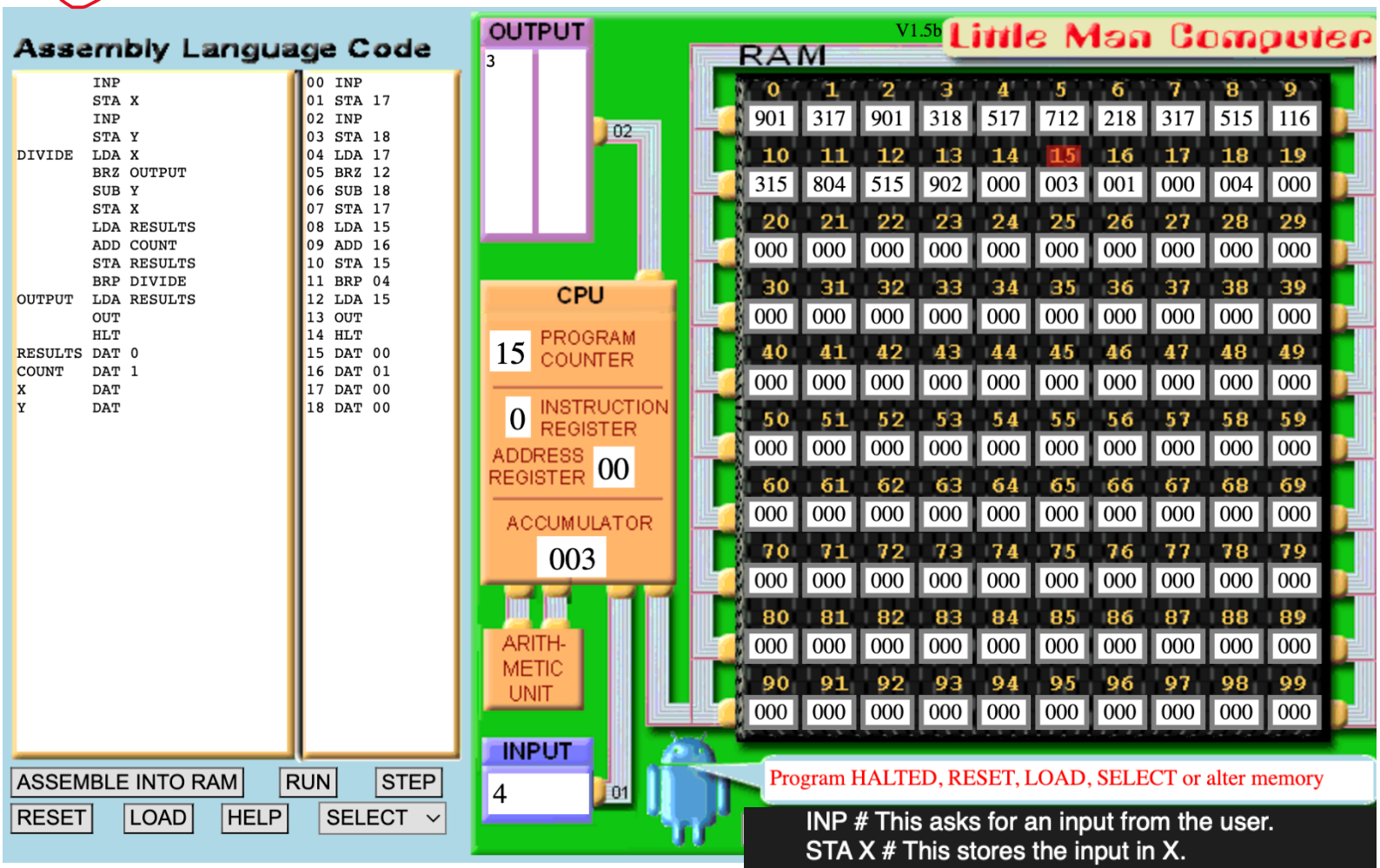
2^3 = 8

This gets an input. It stores it in X. The value in RESULT doubles for X amount of times because exponentiation in LMC is done by repeating addition. Every iteration, X is subtracted by 1. If X becomes 0, it jumps to OUTPUT and outputs the value in RESULT

```
      INP # This asks for an input.
      STA X # This stores it in X.
      LDA COUNT # Loads the value COUNT which was 1.
      STA RESULT # This stores value of COUNT in RESULT.
INDICE  LDA X # This loads the value from X.
      BRZ OUTPUT # If X is 0 it jumps to OUTPUT.
      LDA RESULT # This loads the value from RESULT.
      ADD RESULT # This adds the value from RESULT. So it doubles it.
      STA RESULT #  This stores the value into RESULT.
      LDA X # This loads the value of X.
      SUB COUNT # This subtracts COUNT by 1.
      STA X # This loads the value of X.
      BRP INDICE # If X is positive, it will jump to INDICE.
OUTPUT  LDA RESULT # This loads RESULT.
      OUT # This outputs the value from RESULT.
      HLT # This halts the program.

# This stores 0 in RESULTS at the start.
# This stores 1 in COUNT at the start.
# This creates space for X.
RESULT  DAT 0
COUNT   DAT 1
X       DAT
```

3

X/Y



This asks the user to input two numbers and divides them by each other. There is no division in LMC assembly language, so the program repeatedly subtracts Y from X. It counts how many times Y goes into X. Once a subtraction iteration occurs, it subtracts 1 from X and if X is 0, it jumps to the OUTPUT and returns how many times Y could be in the input of X. Then it halts the program.

12 / 4 = 3

```
      INP # This asks for an input from the user.
      STA X # This stores the input in X.
      INP # This asks for an input from the user.
      STA Y # This asks for an input
DIVIDE  LDA X # This loads X.
      BRZ OUTPUT # If it is 0, it will jump to OUTPUT.
      SUB Y # It subtracts X - Y.
      STA X # It stores it in X.
      LDA RESULTS # It loads RESULTS.
      ADD COUNT # It Adds 1 to COUNT.
      STA RESULTS # It stores it in RESULTS.
      BRP DIVIDE # If it is positive it jumps to DIVIDE.
OUTPUT  LDA RESULTS # This loads RESULTS.
      OUT # This will output RESULTS.
      HLT # This halts the program.

# This stores 0 in RESULTS.
# This stores 1 in COUNT.
# This creates space for X.
# This creates space for Y.
RESULTS DAT 0
COUNT   DAT 1
X       DAT
Y       DAT
```