ASSEMBLER IN PYTHON:

```
import os
import copy

path_dir = os.path.dirname(os.path.abspath(__file__))  # directory of the
folder from Python script
##########################################################################
#########################
name_file = 'enter file name here'
##########################################################################
#########################
asm_path = os.path.join(path_dir, f"{name_file}.asm")
bin_path = os.path.join(path_dir, f"{name_file}.hack")

# Symbolic: dest = comp; jump
# Binary:
# acccccc
comp_dict = {
   '0': '0101010', '1': '0111111', '-1': '0111010', 'D': '0001100',
   'A': '0110000', 'M': '1110000', '!D': '0001101', '!A': '0110001',
   '!M': '1110001', '-D': '0001111', '-A': '0110011', '-M': '1110011',
   'D+1': '0011111', 'A+1': '0110111', 'M+1': '1110111', 'D-1': '0001110',
   'A-1': '0110010', 'M-1': '1110010', 'D+A': '0000010', 'D+M': '1000010',
   'D-A': '0010011', 'D-M': '1010011', 'A-D': '0000111', 'M-D': '1000111',
   'D&A': '0000000', 'D&M': '1000000', 'D|A': '0010101', 'D|M': '1010101'
}

# ddd
dest_dict = {
   'null': '000', 'M': '001', 'D': '010', 'MD': '011',
   'A': '100', 'AM': '101', 'AD': '110', 'AMD': '111'
}

# jjj
jump_dict = {
   'null': '000', 'JGT': '001', 'JEQ': '010', 'JGE': '011',
   'JLT': '100', 'JNE': '101', 'JLE': '110', 'JMP': '111'
}

def load_asm() -> list:
```

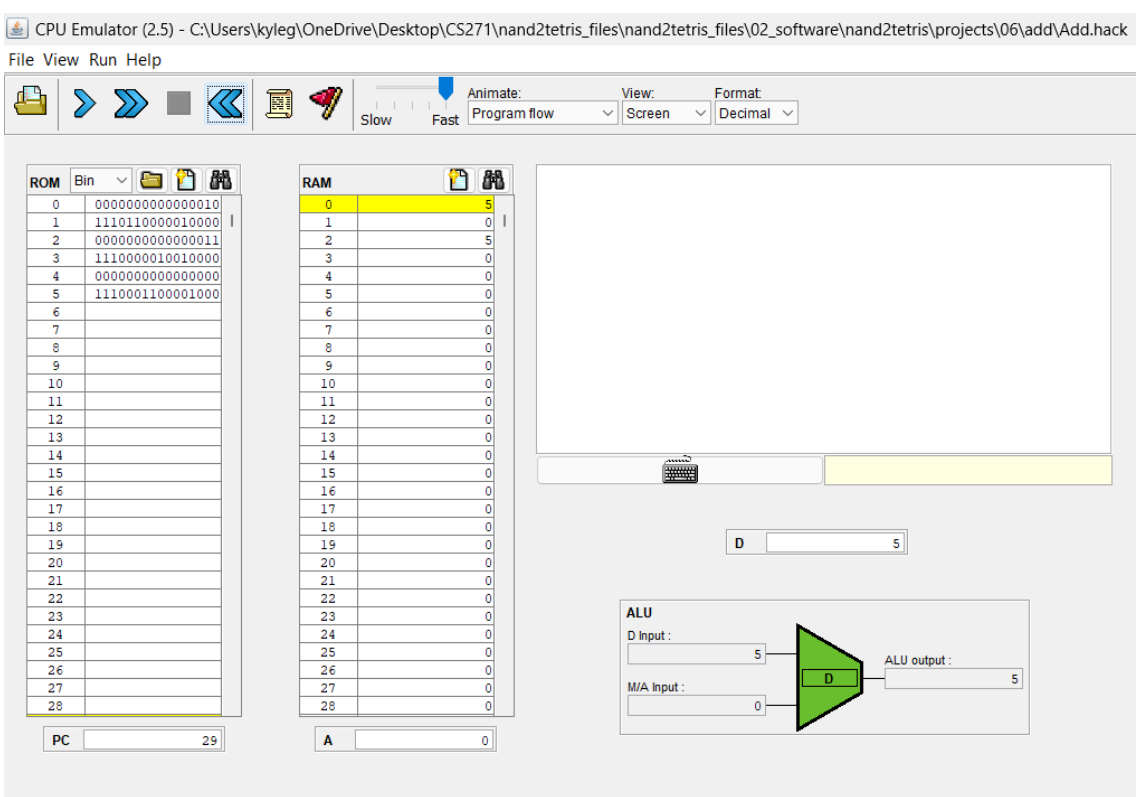
```
    """
    loads the assembly code and returns the code as filtered with no \n or
comments
    """
    filtered_code = []
    with open(asm_path, 'r') as asm:
        for line_no, line in enumerate(asm):
            # Removes \n and comments
            #filtered code is organized as a tuple where line[0] is line #
amd line[1] is code
            line = line.split('//')[0].strip()
            if line:
                filtered_code.append([line_no, line])
    return filtered_code

def symbol_table(code: list) -> dict:
    """
    goes through filtered code and creating symbol table
    """
    symbol = {
     'R0': 0, 'R1': 1, 'R2': 2, 'R3': 3, 'R4': 4, 'R5': 5, 'R6': 6,
     'R7': 7, 'R8': 8, 'R9': 9, 'R10': 10, 'R11': 11, 'R12': 12,
     'R13': 13, 'R14': 14, 'R15': 15, 'SCREEN': 16384, 'KBD': 24576,
     'SP': 0, 'LCL': 1, 'ARG': 2, 'THIS': 3, 'THAT': 4
    }

    instruction_address = 0
    for line in code:
       #looking for loop names which start ( and end with )
       if line[1].startswith('(') and line[1].endswith(')'):
          label = line[1][1:-1]   #removes ()
          if label not in symbol:
             symbol[label] = instruction_address #adds symbol to symbol
table
       else:
          instruction_address += 1 #goes to next memory spot
    #starts at 16 because 0-15 has predefined variables
    next_variable_address = 16
    for line in code:
       if line[1].startswith('@'):
```

```
        symbol_name = line[1][1:]  #removing @
        #adds variable to symbol table by making sure its actually a
variable and not memory address
        if not symbol_name.isdigit() and symbol_name not in symbol: #if
not an integer or found in symbol table
            symbol[symbol_name] = next_variable_address
            next_variable_address += 1
    return symbol

def binary_code(code: list, symbol: dict):
    """
    generates binary code and writes it to .hack file
    """
    with open(bin_path, 'w') as bin_:
        #found this function online to generate twos-complement from base
10
        twos_complement = lambda num, bits: bin((num + (1 << bits)) % (1 <<
bits))[2:].zfill(bits)
        for line in code:
            #don't need to convert named loops again
            if not line[1] or line[1].startswith('('):
                continue
            elif line[1].startswith('@'):
                symbol_name = line[1][1:]
                #gets base 10 value that needs to be converted to binary
                #checks to see if first the symbol is a variable or
directly an integer already
                address = int(symbol[symbol_name]) if symbol_name in symbol
else int(symbol_name)
                #converts to 16 bit two's complement
                byt = twos_complement(address, 16)
                #writes to file
                bin_.write(f'{byt}\n')
            #labeling c-instruction
            else:
                if '=' in line[1]:
                    dest, remainder = line[1].split('=')
                else:
                    dest, remainder = 'null', line[1]
                if ';' in remainder:
```

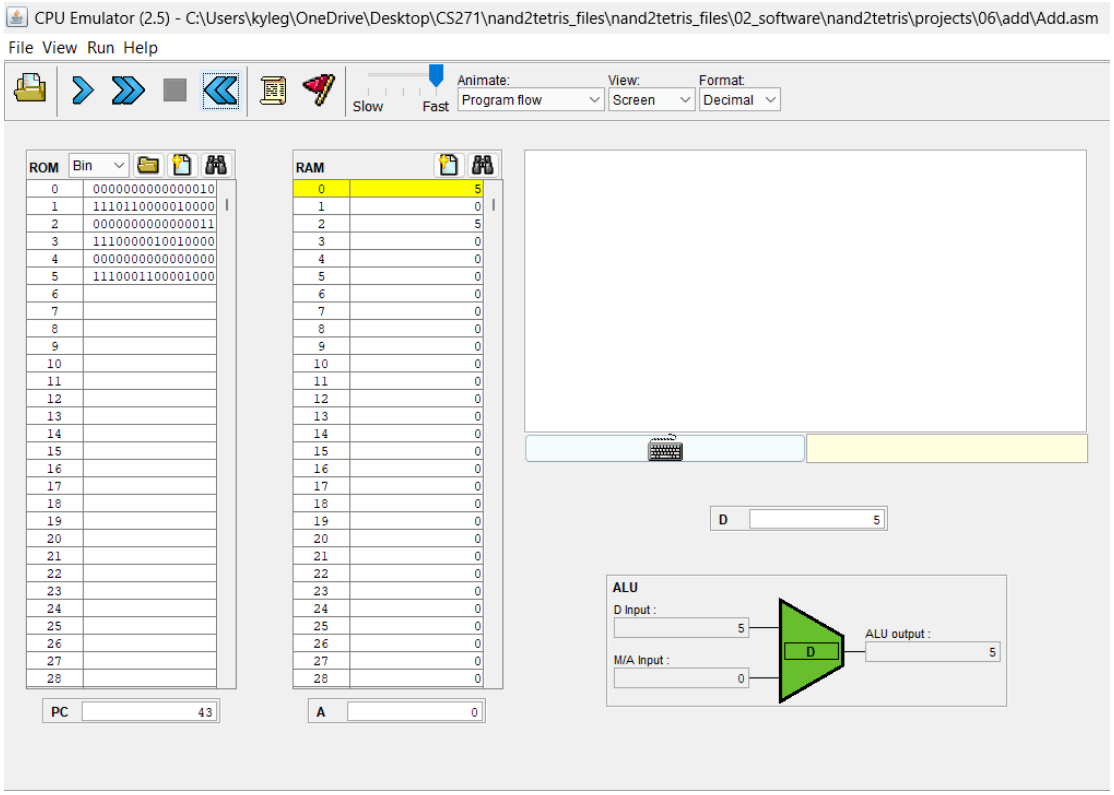
```
                    comp, jump = remainder.split(';')
                else:
                    comp, jump = remainder, 'null'

                dest_1_code = dest_dict[dest]
                comp_1_code = comp_dict[comp]
                jump_1_code = jump_dict[jump]

                bin_code = '111' + comp_1_code + dest_1_code + jump_1_code
                bin_.write(f'{bin_code}\n')

if __name__ == '__main__':
    old_code = load_asm()
    code = copy.deepcopy(old_code)
    symbol = symbol_table(code)
    binary_code(old_code, symbol)
```

To test the assembler, each assembly code in Project 6 was converted into binary/hack code. To fully test the code, I ran the assembly and hack codes and saw if the two results agreed.

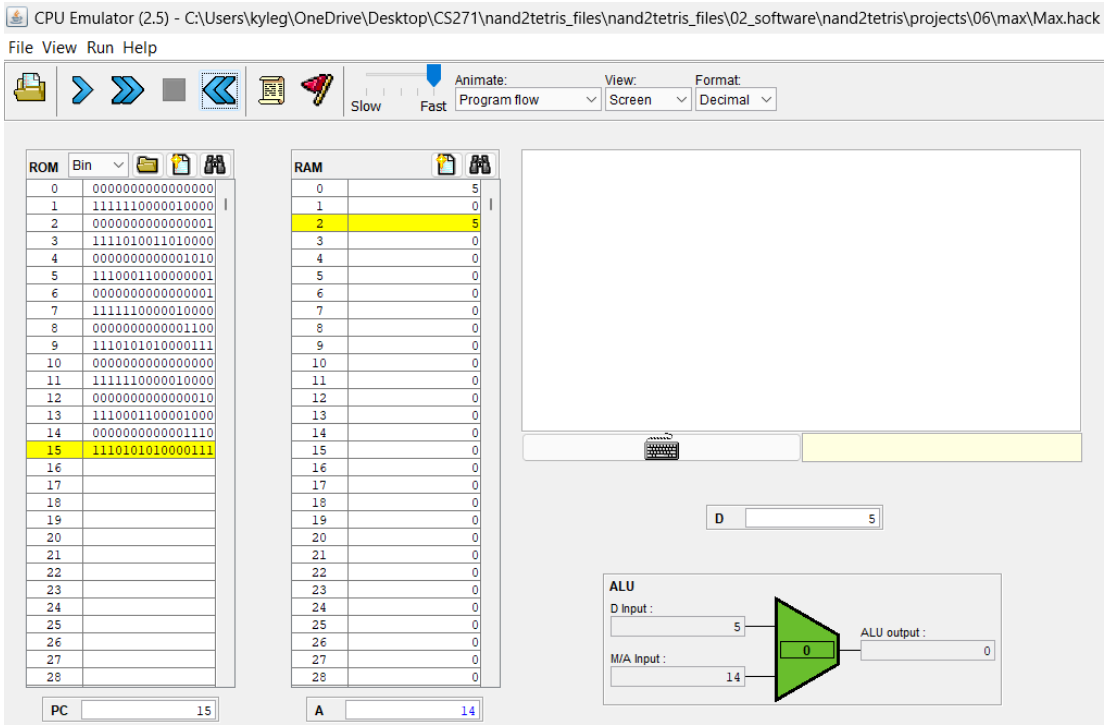
RESULTS OF Add.hack:



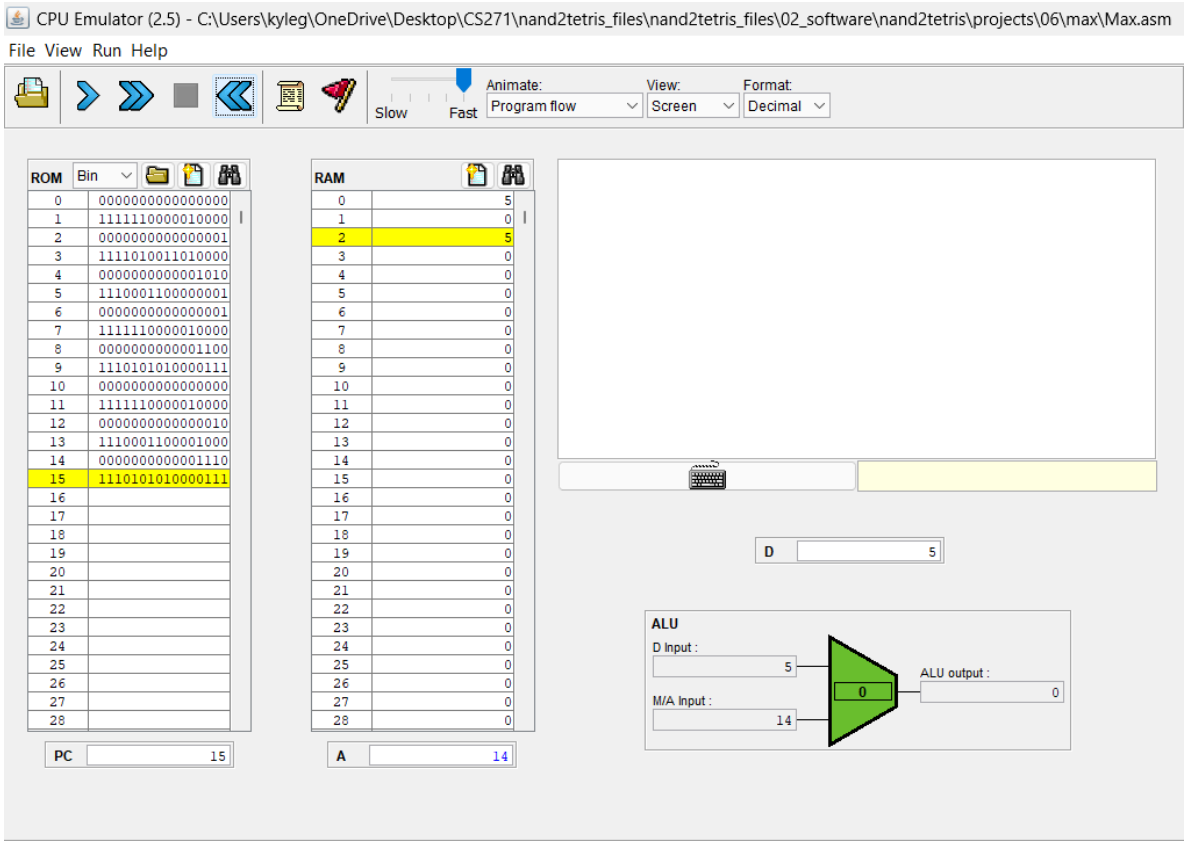
RESULTS OF Add.asm:



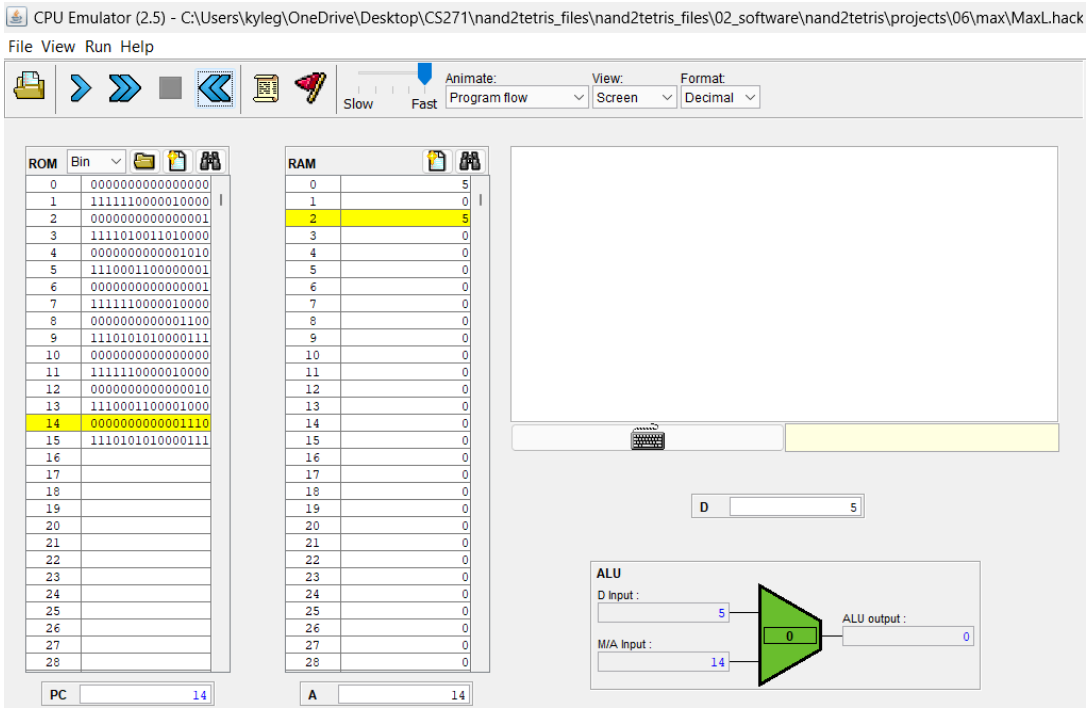
RESULTS OF Max.hack:



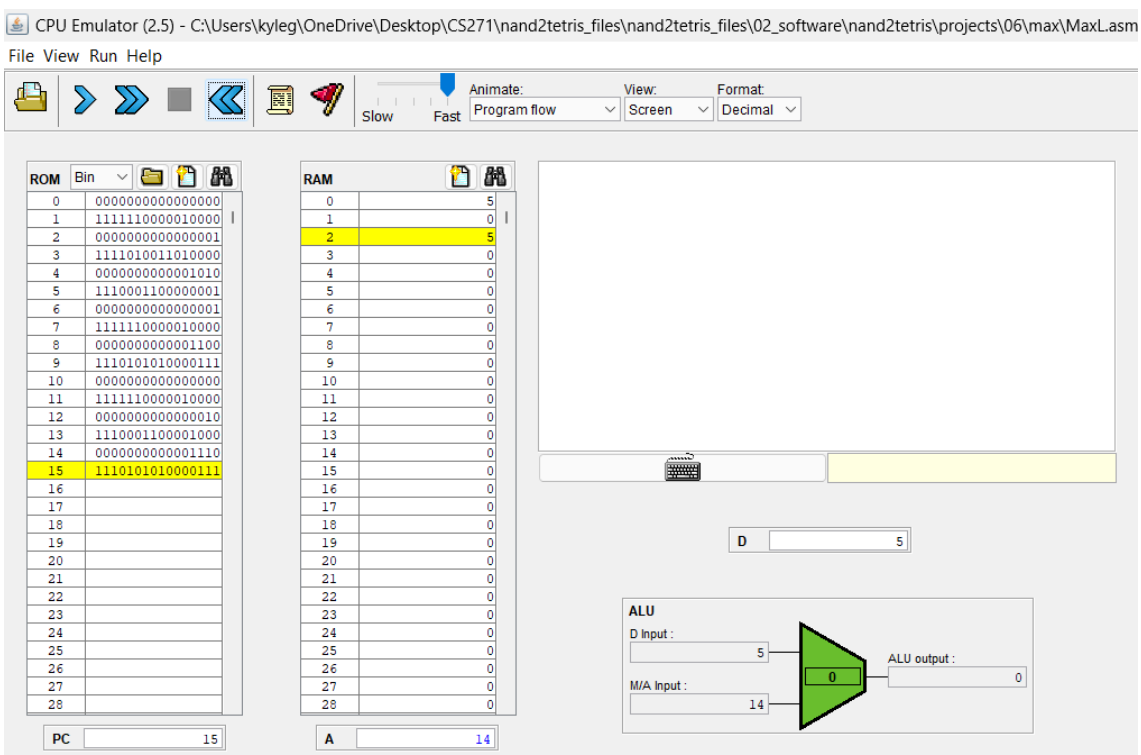
Results of Max.asm:



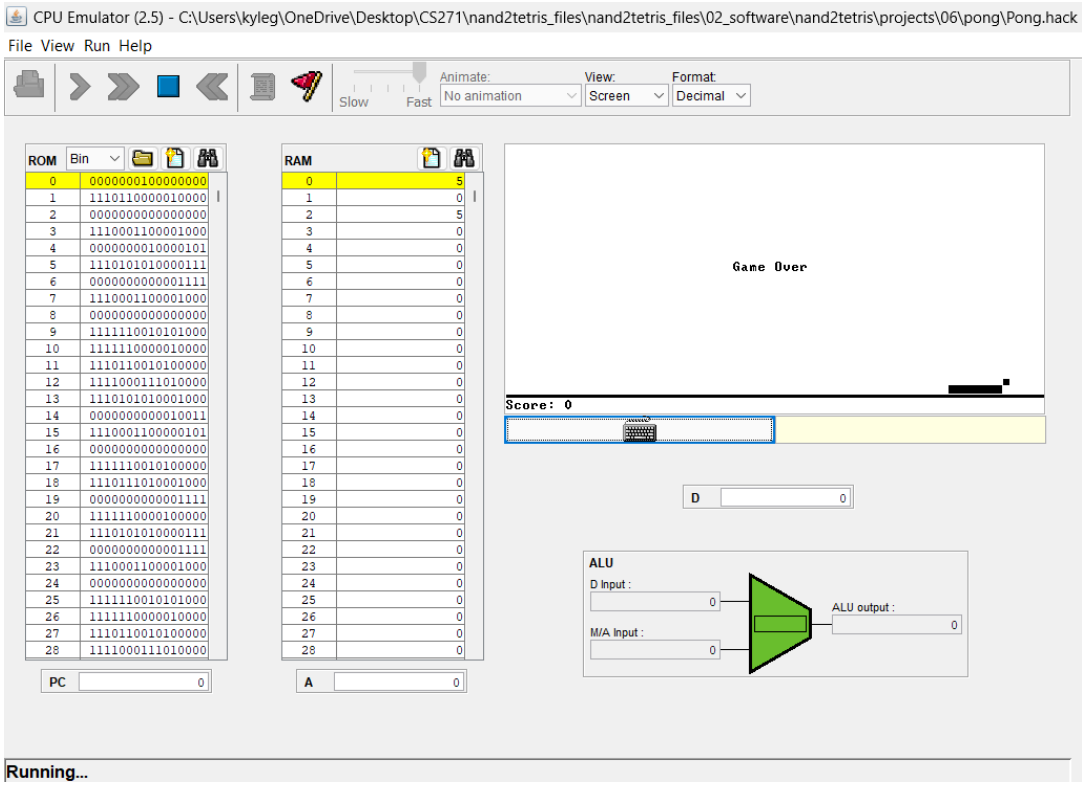
RESULTS OF MaxL.hack:



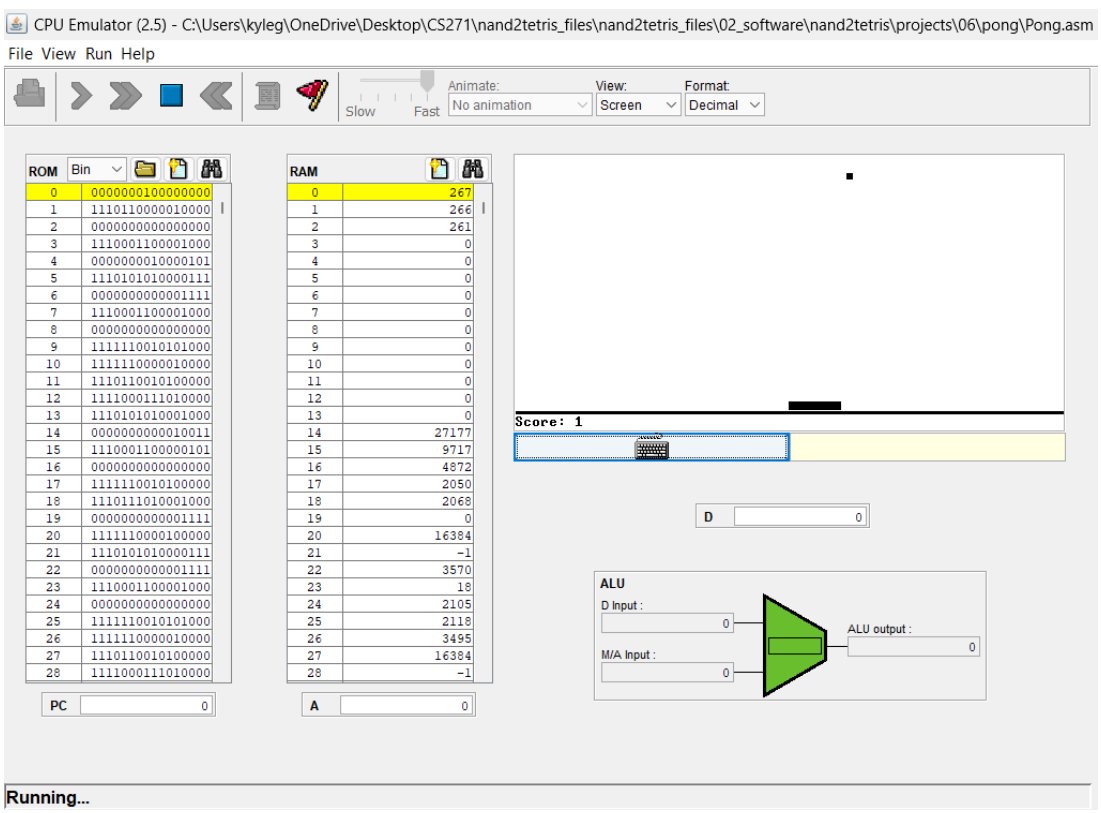
RESULTS OF MaxL.asm:



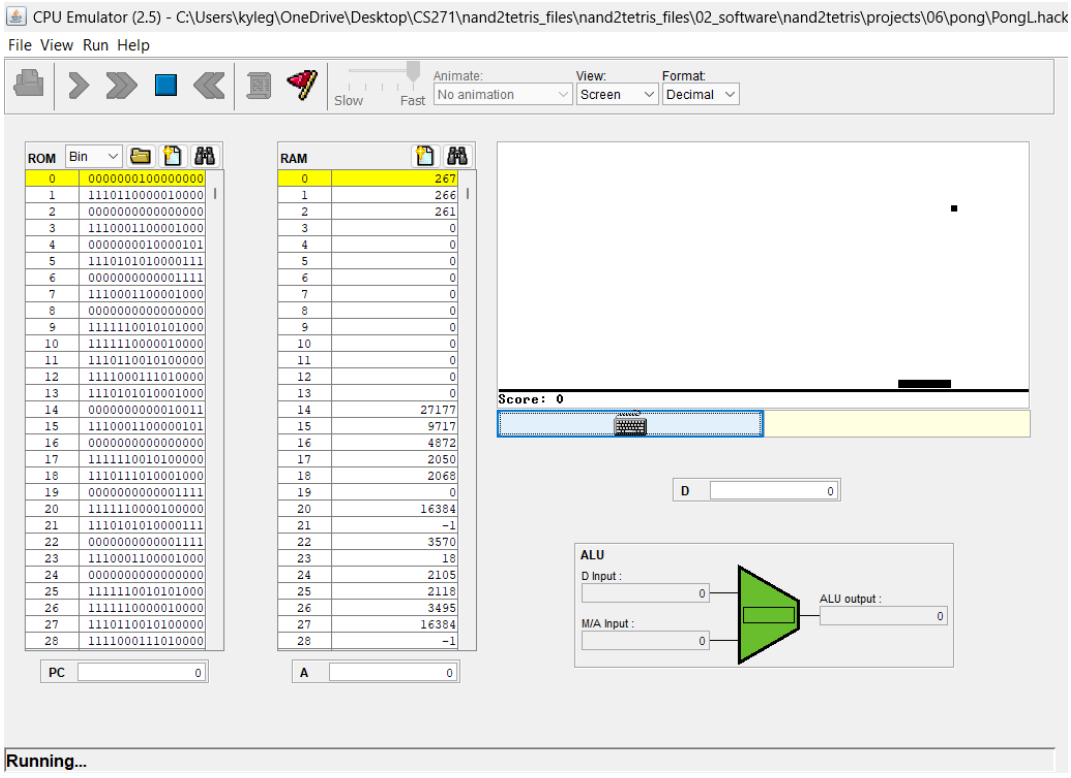
RESULTS OF Pong.hack:



RESULTS OF Pong.asm:

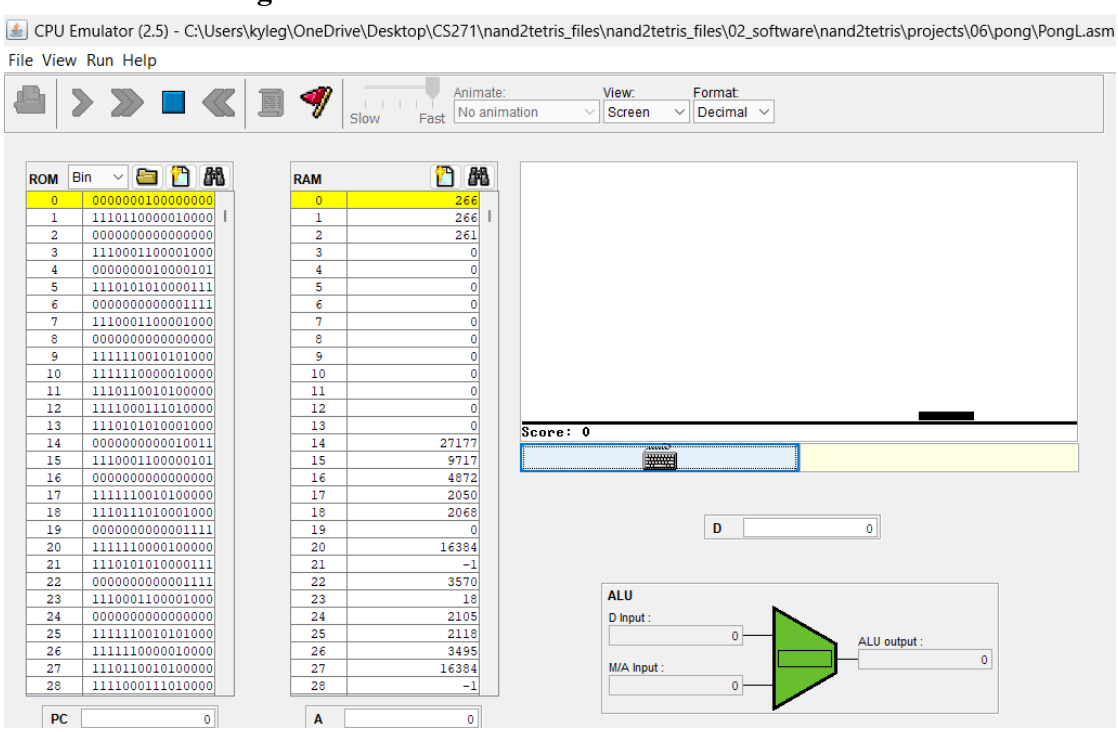


RESULTS OF PongL.hack:



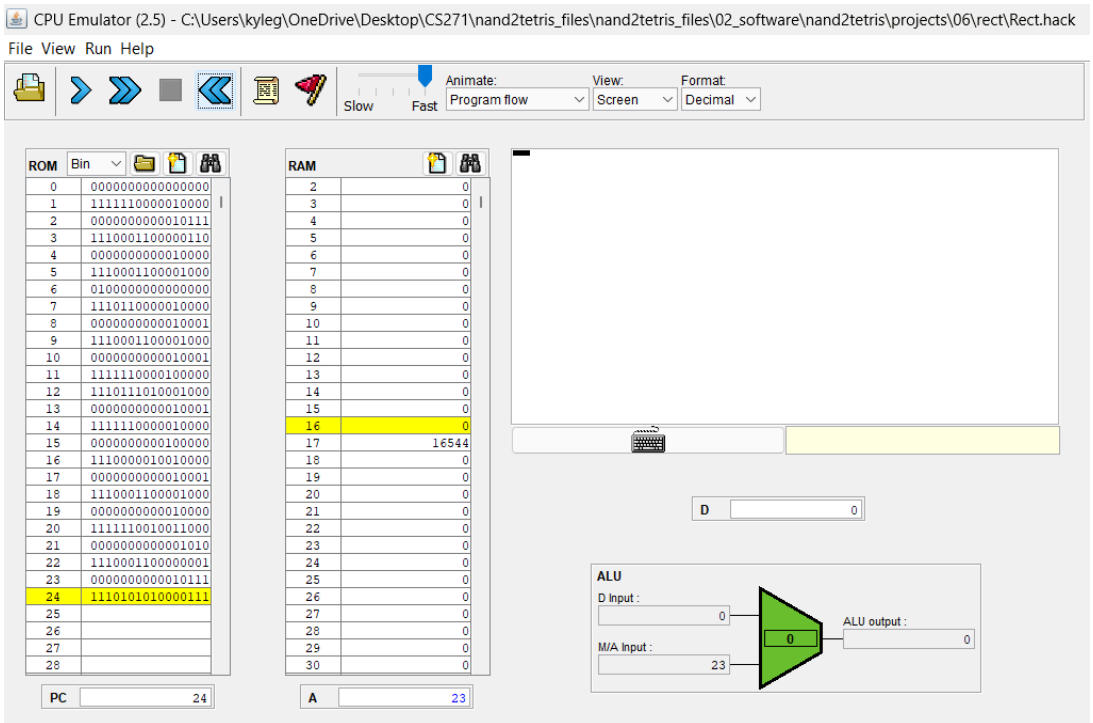
	ROM	RAM
0	0000000100000000	267
1	1110110000010000	266
2	0000000000000000	261
3	1110001100001000	0
4	0000000010000101	0
5	1110101010000111	0
6	0000000000001111	0
7	1110001100001000	0
8	0000000000000000	0
9	1111110010101000	0
10	1111110000010000	0
11	1110110010100000	0
12	1111000111010000	0
13	1110101010001000	0
14	0000000000010011	27177
15	1110001100000101	9717
16	0000000000000000	4872
17	1111110010100000	2050
18	1110111010001000	2068
19	0000000000001111	0
20	1111110000100000	16384
21	1110101010000111	-1
22	0000000000001111	3570
23	1110001100001000	18
24	0000000000000000	2105
25	1111110010101000	2118
26	1111110000010000	3495
27	1110110010100000	16384
28	1111000111010000	-1

RESULTS OF PongL.asm:

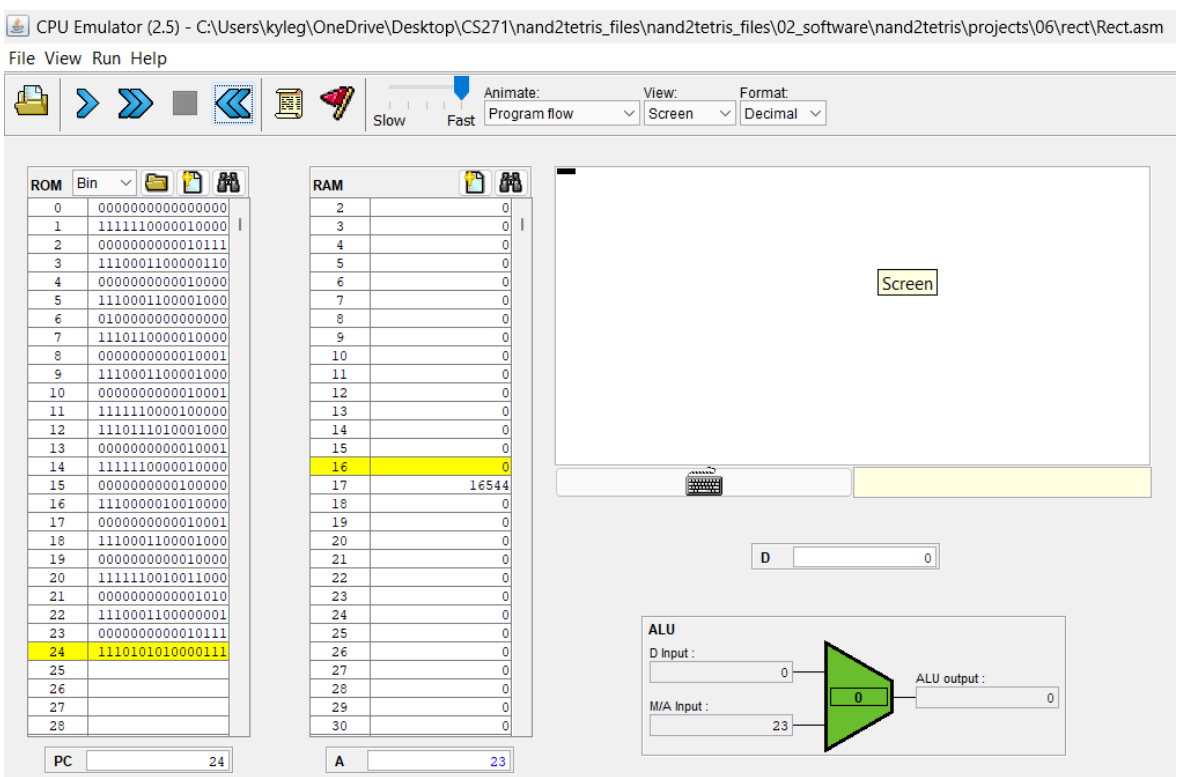


	ROM	RAM
0	0000000100000000	266
1	1110110000010000	266
2	0000000000000000	261
3	1110001100001000	0
4	0000000010000101	0
5	1110101010000111	0
6	0000000000001111	0
7	1110001100001000	0
8	0000000000000000	0
9	1111110010101000	0
10	1111110000010000	0
11	1110110010100000	0
12	1111000111010000	0
13	1110101010001000	0
14	0000000000010011	27177
15	1110001100000101	9717
16	0000000000000000	4872
17	1111110010100000	2050
18	1110111010001000	2068
19	0000000000001111	0
20	1111110000100000	16384
21	1110101010000111	-1
22	0000000000001111	3570
23	1110001100001000	18
24	0000000000000000	2105
25	1111110010101000	2118
26	1111110000010000	3495
27	1110110010100000	16384
28	1111000111010000	-1

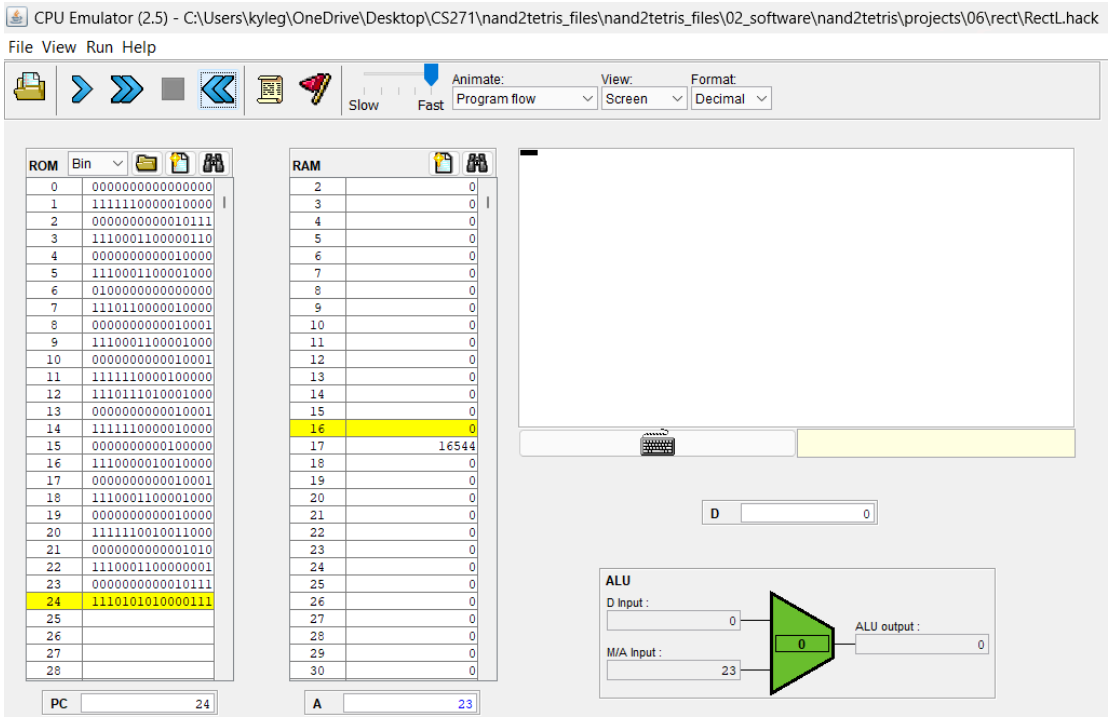
RESULTS of Rect.hack:



RESULTS of rect.asm:



RESULTS of RectL.hack:



RESULTS of RectL.asm: