Computer Architecture
Assignment – 1
IAS processor design

Armstrong number

KNV Aditya - IMT2023033
Abhijit Dibbidi - IMT2023054
Kusumanchi Vinay - IMT2023608

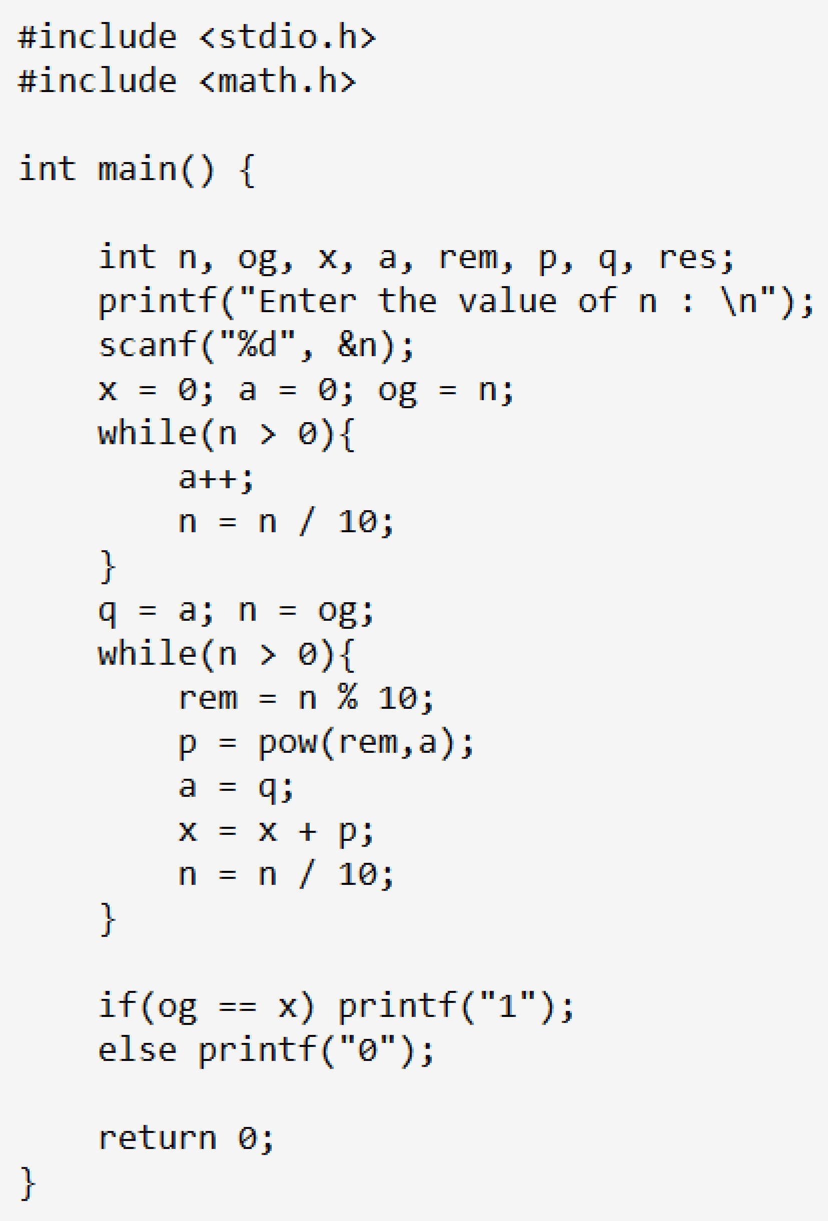
Armstrong numbers:

An Armstrong number is a number that is the sum of its own digits, each raised to the power of the number of digits.

Let us take a number abc... where $a,b,c,\ldots$ are its individual digits. If the number is a n-digit number, it is an Armstrong number if:
$a\text{^}n+b\text{^}n+c\text{^}n+\ldots=abc\ldots$

For example, let's take the number 153:
$1\text{^}3+5\text{^}3+3\text{^}3=1+125+27=153$
So, 153 is an Armstrong number.

The C code we wrote to check if a given number is an Armstrong number or not is:



```c
#include <stdio.h>
#include <math.h>

int main() {

    int n, og, x, a, rem, p, q, res;
    printf("Enter the value of n : \n");
    scanf("%d", &n);
    x = 0; a = 0; og = n;
    while(n > 0){
        a++;
        n = n / 10;
    }
    q = a; n = og;
    while(n > 0){
        rem = n % 10;
        p = pow(rem,a);
        a = q;
        x = x + p;
        n = n / 10;
    }

    if(og == x) printf("1");
    else printf("0");

    return 0;
}
```

The code includes the math library to use the ‘pow’ function.
The variables used are:

n : Input number

og : Original number

x : Accumulator for the sum of digits raised to the power of the number of digits

a : Variable to count the number of digits

rem : Variable to store the remainder when dividing by 10

p : Variable to store the power of each digit

q : Variable to store the original value of ‘a’

res : Result variable

Firstly, the code asks for an input from the user and stores it in ‘n’. Then it initializes the variables ‘x’, ‘a’ and stores the original input in og.

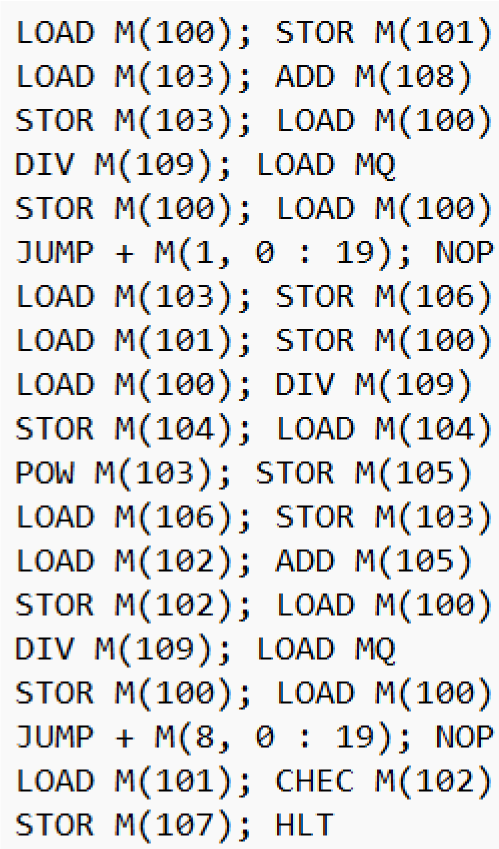
Then it counts the number of digits in the input number using a while loop which divides ‘n’ by 10 in each iteration until ‘n’ becomes 0, incrementing ‘a’ for each division.

Then it stores the original value of ‘a’ in ’q’ and resets ’n’ to the original input number.

Then the code calculates the sum of each digit raised to the power of the number of digits using the ’pow’ function. The result is stored in the variable ’x’.

Then it compares the original input number (og) with the calculated sum (x). If they are equal, it means the input is an Armstrong number, and ’res’ is assigned the value 1; otherwise, it is assigned the value 0.

The IAS Assembly program for the above C program is:



```
LOAD M(100); STOR M(101)
LOAD M(103); ADD M(108)
STOR M(103); LOAD M(100)
DIV M(109); LOAD MQ
STOR M(100); LOAD M(100)
JUMP + M(1, 0 : 19); NOP
LOAD M(103); STOR M(106)
LOAD M(101); STOR M(100)
LOAD M(100); DIV M(109)
STOR M(104); LOAD M(104)
POW M(103); STOR M(105)
LOAD M(106); STOR M(103)
LOAD M(102); ADD M(105)
STOR M(102); LOAD M(100)
DIV M(109); LOAD MQ
STOR M(100); LOAD M(100)
JUMP + M(8, 0 : 19); NOP
LOAD M(101); CHEC M(102)
STOR M(107); HLT
```

We assigned random addresses to the variables and constants:

n = 100
og = 101
x = 102
a = 103
rem = 104
p = 105
q = 106
res = 107
1 = 108
10 = 109

In the above assembly program,

`LOAD` loads a value from memory into a register

`STOR` stores the content of a register into memory.

`ADD` performs addition.

`DIV` performs division, storing the quotient in the MQ register.

`JUMP` is a conditional jump instruction.

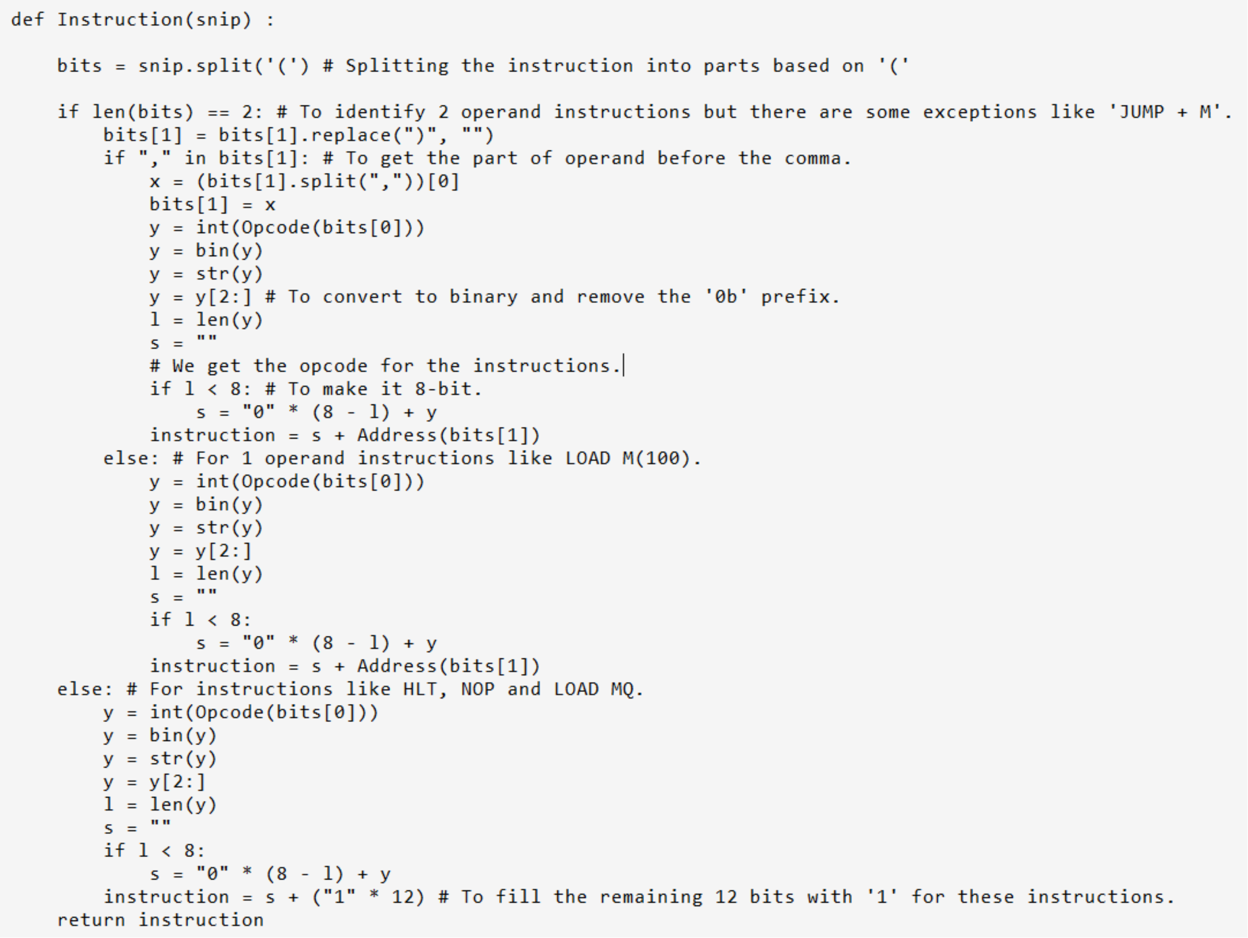
`NOP` is a no-operation instruction.

`POW` is a new instruction introduced by us to perform the power operation.

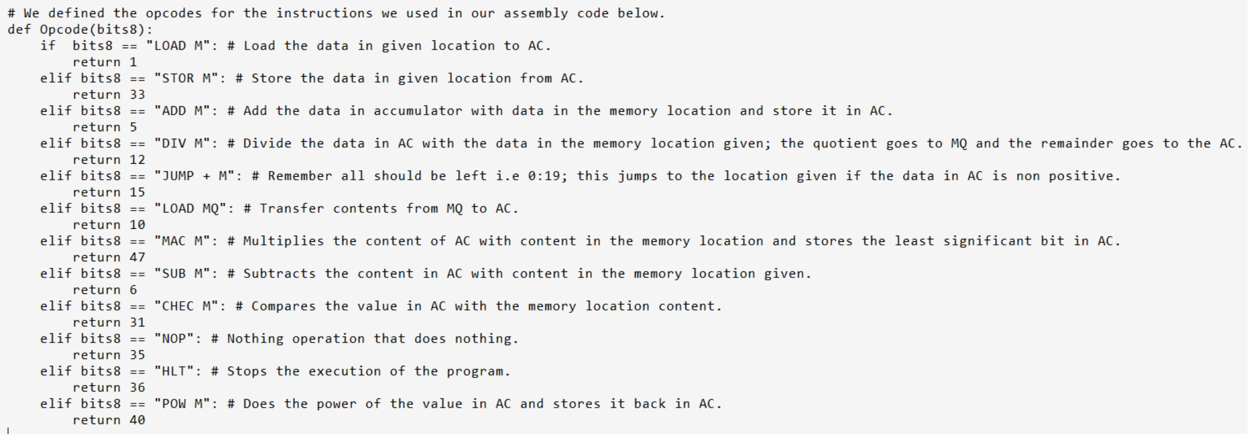
`CHEC` is another new instruction introduced by us to perform the conditional check operation.

`HLT` indicates the end of the program.

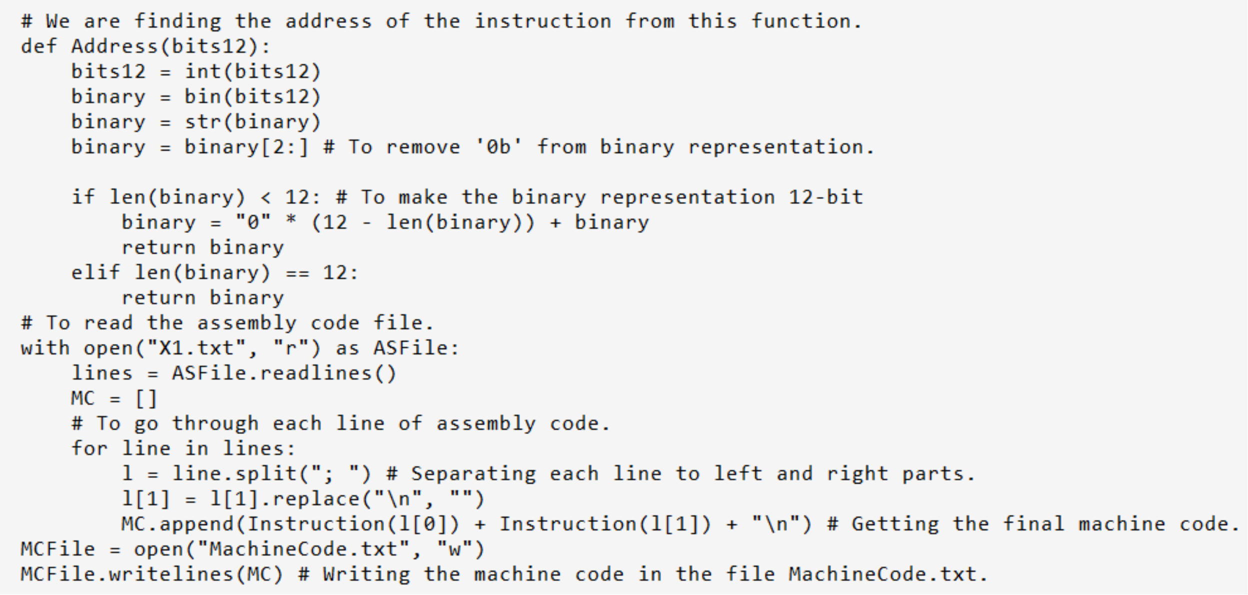
The assembler code for the above assembly code with comments explaining the process:



```
def Instruction(snip) :

    bits = snip.split('(') # Splitting the instruction into parts based on '('

    if len(bits) == 2: # To identify 2 operand instructions but there are some exceptions like 'JUMP + M'.
        bits[1] = bits[1].replace(")", "")
        if "," in bits[1]: # To get the part of operand before the comma.
            x = (bits[1].split(","))[0]
            bits[1] = x
            y = int(Opcode(bits[0]))
            y = bin(y)
            y = str(y)
            y = y[2:] # To convert to binary and remove the '0b' prefix.
            l = len(y)
            s = ""
            # We get the opcode for the instructions.
            if l < 8: # To make it 8-bit.
                s = "0" * (8 - l) + y
            instruction = s + Address(bits[1])
        else: # For 1 operand instructions like LOAD M(100).
            y = int(Opcode(bits[0]))
            y = bin(y)
            y = str(y)
            y = y[2:]
            l = len(y)
            s = ""
            if l < 8:
                s = "0" * (8 - l) + y
            instruction = s + Address(bits[1])
    else: # For instructions like HLT, NOP and LOAD MQ.
        y = int(Opcode(bits[0]))
        y = bin(y)
        y = str(y)
        y = y[2:]
        l = len(y)
        s = ""
        if l < 8:
            s = "0" * (8 - l) + y
        instruction = s + ("1" * 12) # To fill the remaining 12 bits with '1' for these instructions.
    return instruction
```

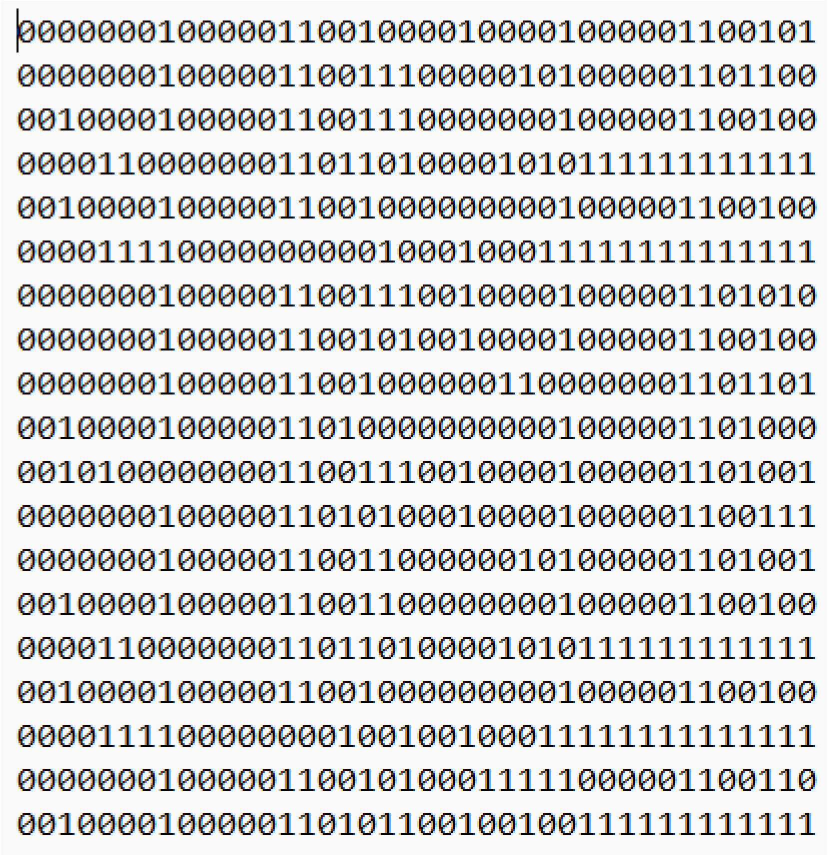


```
# We defined the opcodes for the instructions we used in our assembly code below.
def Opcode(bits8):
    if  bits8 == "LOAD M": # Load the data in given location to AC.
        return 1
    elif bits8 == "STOR M": # Store the data in given location from AC.
        return 33
    elif bits8 == "ADD M": # Add the data in accumulator with data in the memory location and store it in AC.
        return 5
    elif bits8 == "DIV M": # Divide the data in AC with the data in the memory location given; the quotient goes to MQ and the remainder goes to the AC.
        return 12
    elif bits8 == "JUMP + M": # Remember all should be left i.e 0:19; this jumps to the location given if the data in AC is non positive.
        return 15
    elif bits8 == "LOAD MQ": # Transfer contents from MQ to AC.
        return 10
    elif bits8 == "MAC M": # Multiplies the content of AC with content in the memory location and stores the least significant bit in AC.
        return 47
    elif bits8 == "SUB M": # Subtracts the content in AC with content in the memory location given.
        return 6
    elif bits8 == "CHEC M": # Compares the value in AC with the memory location content.
        return 31
    elif bits8 == "NOP": # Nothing operation that does nothing.
        return 35
    elif bits8 == "HLT": # Stops the execution of the program.
        return 36
    elif bits8 == "POW M": # Does the power of the value in AC and stores it back in AC.
        return 40
```



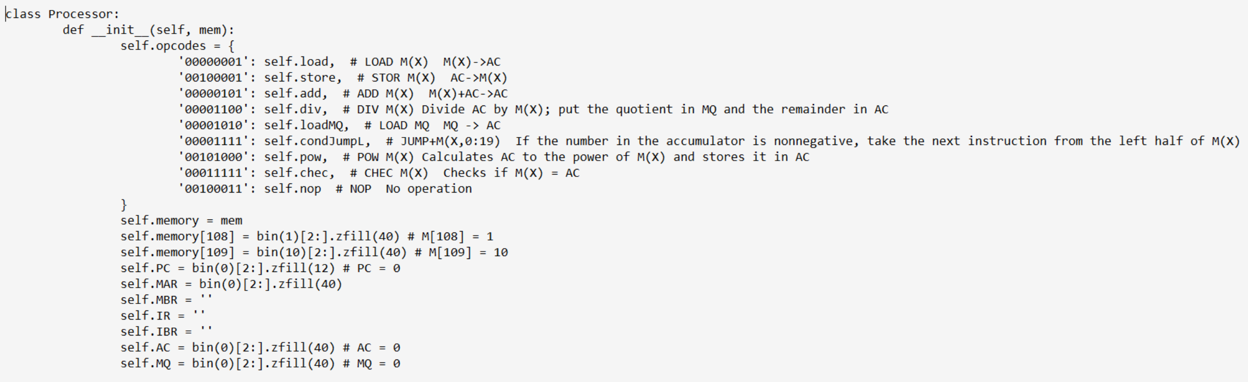
```
# We are finding the address of the instruction from this function.
def Address(bits12):
    bits12 = int(bits12)
    binary = bin(bits12)
    binary = str(binary)
    binary = binary[2:] # To remove '0b' from binary representation.

    if len(binary) < 12: # To make the binary representation 12-bit
        binary = "0" * (12 - len(binary)) + binary
        return binary
    elif len(binary) == 12:
        return binary
# To read the assembly code file.
with open("X1.txt", "r") as ASFile:
    lines = ASFile.readlines()
    MC = []
    # To go through each line of assembly code.
    for line in lines:
        l = line.split("; ") # Separating each line to left and right parts.
        l[1] = l[1].replace("\n", "")
        MC.append(Instruction(l[0]) + Instruction(l[1]) + "\n") # Getting the final machine code.
MCFile = open("MachineCode.txt", "w")
MCFile.writelines(MC) # Writing the machine code in the file MachineCode.txt.
```

Machine code produced by using the above assembler program:

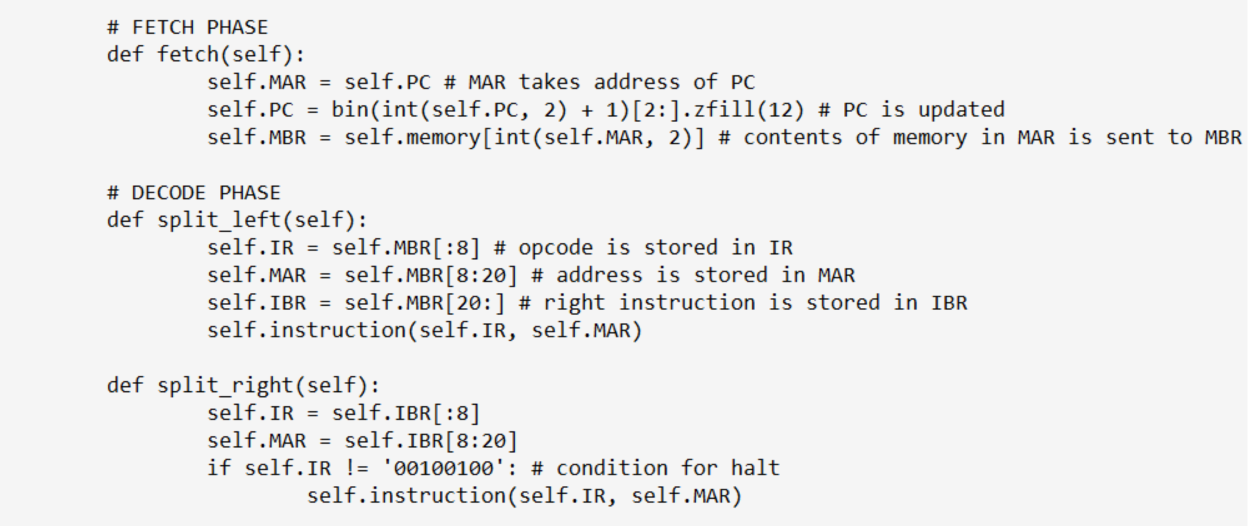


```
0000000100000110010000100001000001100101
0000000100000110011100000101000001101100
0010000100000110011100000001000001100100
0000110000000110110100001010111111111111
0010000100000110010000000001000001100100
0000111000000000001000100011111111111111
0000000100000110011100100001000001101010
0000000100000110010100100001000001100100
0000000100000110010000001100000001101101
0010000100000110100000000001000001101000
0010100000000110011100100001000001101001
0000000100000110101000100001000001100111
0000000100000110011000000101000001101001
0010000100000110011000000001000001100100
0000110000000110110100001010111111111111
0010000100000110010000000001000001100100
0000111000000000100100100011111111111111
0000000100000110010100011111000001100110
0010000100000110101100100100111111111111
```

This is the processor code for the machine code generated by the above assembler program explained through comments:



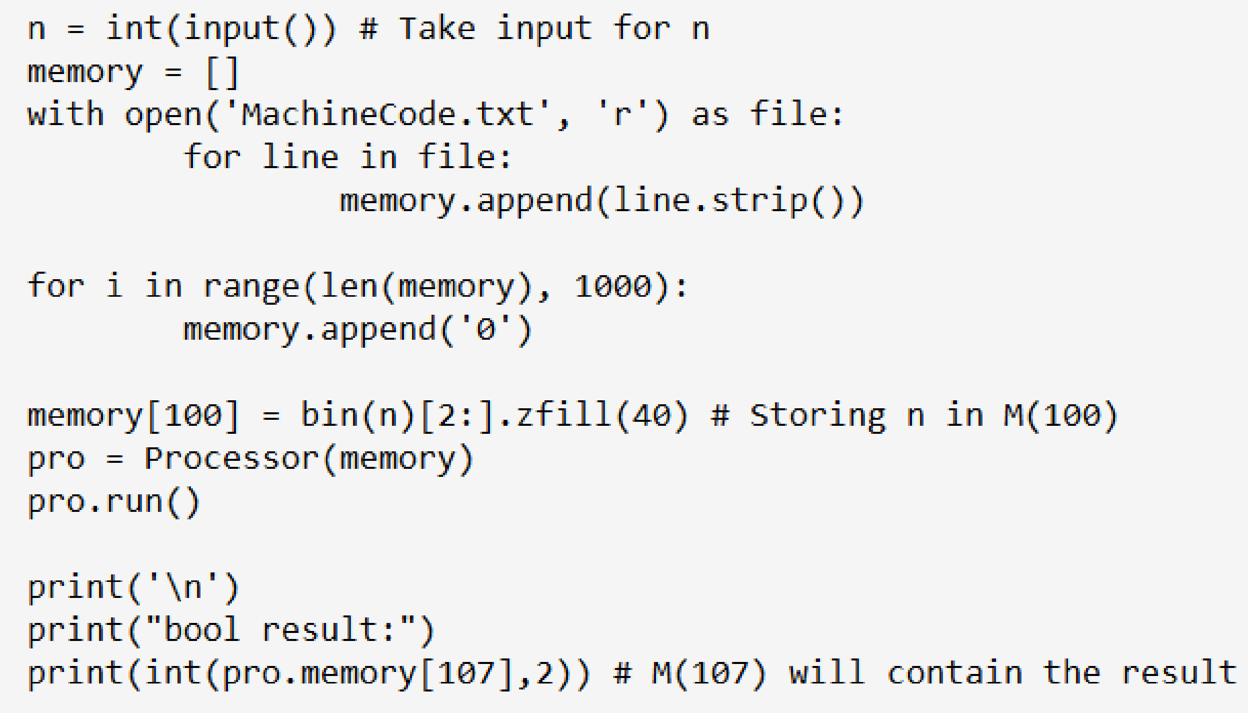
```
class Processor:
        def __init__(self, mem):
                self.opcodes = {
                        '00000001': self.load,  # LOAD M(X)  M(X)->AC
                        '00100001': self.store,  # STOR M(X)  AC->M(X)
                        '00000101': self.add,  # ADD M(X)  M(X)+AC->AC
                        '00001100': self.div,  # DIV M(X) Divide AC by M(X); put the quotient in MQ and the remainder in AC
                        '00001010': self.loadMQ,  # LOAD MQ  MQ -> AC
                        '00001111': self.condJumpL,  # JUMP+M(X,0:19)  If the number in the accumulator is nonnegative, take the next instruction from the left half of M(X)
                        '00101000': self.pow,  # POW M(X) Calculates AC to the power of M(X) and stores it in AC
                        '00011111': self.chec,  # CHEC M(X)  Checks if M(X) = AC
                        '00100011': self.nop  # NOP  No operation
                }
                self.memory = mem
                self.memory[108] = bin(1)[2:].zfill(40) # M[108] = 1
                self.memory[109] = bin(10)[2:].zfill(40) # M[109] = 10
                self.PC = bin(0)[2:].zfill(12) # PC = 0
                self.MAR = bin(0)[2:].zfill(40)
                self.MBR = ''
                self.IR = ''
                self.IBR = ''
                self.AC = bin(0)[2:].zfill(40) # AC = 0
                self.MQ = bin(0)[2:].zfill(40) # MQ = 0
```



```
        # FETCH PHASE
        def fetch(self):
                self.MAR = self.PC # MAR takes address of PC
                self.PC = bin(int(self.PC, 2) + 1)[2:].zfill(12) # PC is updated
                self.MBR = self.memory[int(self.MAR, 2)] # contents of memory in MAR is sent to MBR

        # DECODE PHASE
        def split_left(self):
                self.IR = self.MBR[:8] # opcode is stored in IR
                self.MAR = self.MBR[8:20] # address is stored in MAR
                self.IBR = self.MBR[20:] # right instruction is stored in IBR
                self.instruction(self.IR, self.MAR)

        def split_right(self):
                self.IR = self.IBR[:8]
                self.MAR = self.IBR[8:20]
                if self.IR != '00100100': # condition for halt
                        self.instruction(self.IR, self.MAR)
```

```
    # EXECUTE PHASE
    def instruction(self, opcode, address):
            self.opcodes[opcode](address)

    def load(self,x):  # LOAD M(X)
            self.MBR = self.memory[int(x,2)] # Contents of M(X) is loaded MBR
            self.AC = bin(int(self.MBR,2))[2:].zfill(40) # MBR to AC
            print(f'load m {int(x,2)}')

    def store(self,x):  # STOR M(X)
            self.memory[int(x,2)] = bin(int(self.AC, 2))[2:].zfill(40) # AC is stored in M(X)
            print(f'store m {int(x,2)}')

    def add(self,x):  # ADD M(X)
            self.MBR = self.memory[int(x,2)]
            self.AC = bin(int(self.AC, 2) + int(self.MBR, 2))[2:].zfill(40)
            print(f'add m {int(x,2)}')

    def div(self,x):  # DIV M(X)
            quo = int(self.AC, 2) // int(self.memory[int(x,2)], 2)
            rem = int(self.AC, 2) % int(self.memory[int(x,2)], 2)
            self.AC = bin(rem)[2:].zfill(40) # Put remainder in AC
            self.MQ = bin(quo)[2:].zfill(40) # Quotient in MQ
            print(f'div m {int(x,2)}')

    def loadMQ(self,x):
            self.AC = self.MQ
            print('load mq')

    def condJumpL(self,x):  # JUMP+M(X,0:19)
            if int(self.AC, 2) > 0: # check condition
                    print('jump')
                    self.PC = x # change PC value to loop back to an address
                    self.fetch()
                    self.split_left()

            else: # if condition fails, continue
                    print('break out of loop')
```

```
    def pow(self, x):  # POW M(X)
            self.AC = bin(int(self.AC, 2) ** int(self.memory[int(x,2)], 2))[2:].zfill(40)
            print(f'pow m {int(x,2)}')

    def chec(self,x):  # CHEC M(X)
            print(f'chec m {int(x,2)}')
            if self.AC == self.memory[int(x,2)]: # check if value in accumulator == M(x)
                    self.AC = bin(1)[2:].zfill(40)
            else:
                    self.AC = bin(0)[2:].zfill(40)

    def nop(self,x): # No Operation
            print(f'nop')

    def run(self):
            while self.IR != '00100100': # HLT condition
                    self.fetch()
                    self.split_left()
                    self.split_right()
```



```python
n = int(input()) # Take input for n
memory = []
with open('MachineCode.txt', 'r') as file:
        for line in file:
                memory.append(line.strip())

for i in range(len(memory), 1000):
        memory.append('0')

memory[100] = bin(n)[2:].zfill(40) # Storing n in M(100)
pro = Processor(memory)
pro.run()

print('\n')
print("bool result:")
print(int(pro.memory[107],2)) # M(107) will contain the result
```