IAS Architecture Design

Course Name: EG 212, Computer Architecture - Processor Design

Shubhranil Basak IMT2023510

Gathik Jindal IMT2023089

Introduction:

The IAS machine was the first electronic computer built at the Institute for Advanced Study (IAS) in Princeton, New Jersey. It is sometimes called the von Neumann machine, since the paper describing its design was edited by John von Neumann, a mathematics professor at both Princeton University and IAS. The computer was built under his direction, starting in 1946 and finished in 1951. The general organization is called von Neumann architecture, even though it was both conceived and implemented by others. The computer is in the collection of the Smithsonian National Museum of American History but is not currently on display. [1]

The IAS followed a stored-program architecture. It used about a 1000 memory locations all comprising of vaccuum tubes. It had a central control unit with an arithmetic logic unit (ALU). Its memory system allowed for random access, as well as dynamically changing parts or the entire word itself.

Assembler:

The Assembler, a python program that converts our assembly file to an object file follows a very basic approach. The few extra commands we added are jump++ which jumps if the AC is greater that 0, HALT to halt the program and NOP to signify no code or instruction in that particular memory location or part of that memory location. The assembler acts as a text replacer, and it also ignores all comments. (lines starting with //).

Processor:

The Processor, being a little more complex than the assembler, follows the flow chart depicted below. It has a variety of classes owing to the different objects and their requirements. Obviously this could have been more modularized but just for the sake of a project on the IAS ISA it seemed redundant. We have implemented the Main Memory, the IBR, the MAR and the CPU itself as classes with their respective methods. One instruction was giving us a hard time was the jump right instruction. Since the IAS machine only goes to the next instruction when IBR is empty, and IBR is only empty when both instructions are executed, hence we had to set a flag variable that would let the processor know that it only has to execute the right instruction and not both the left and the right instruction at the same time.

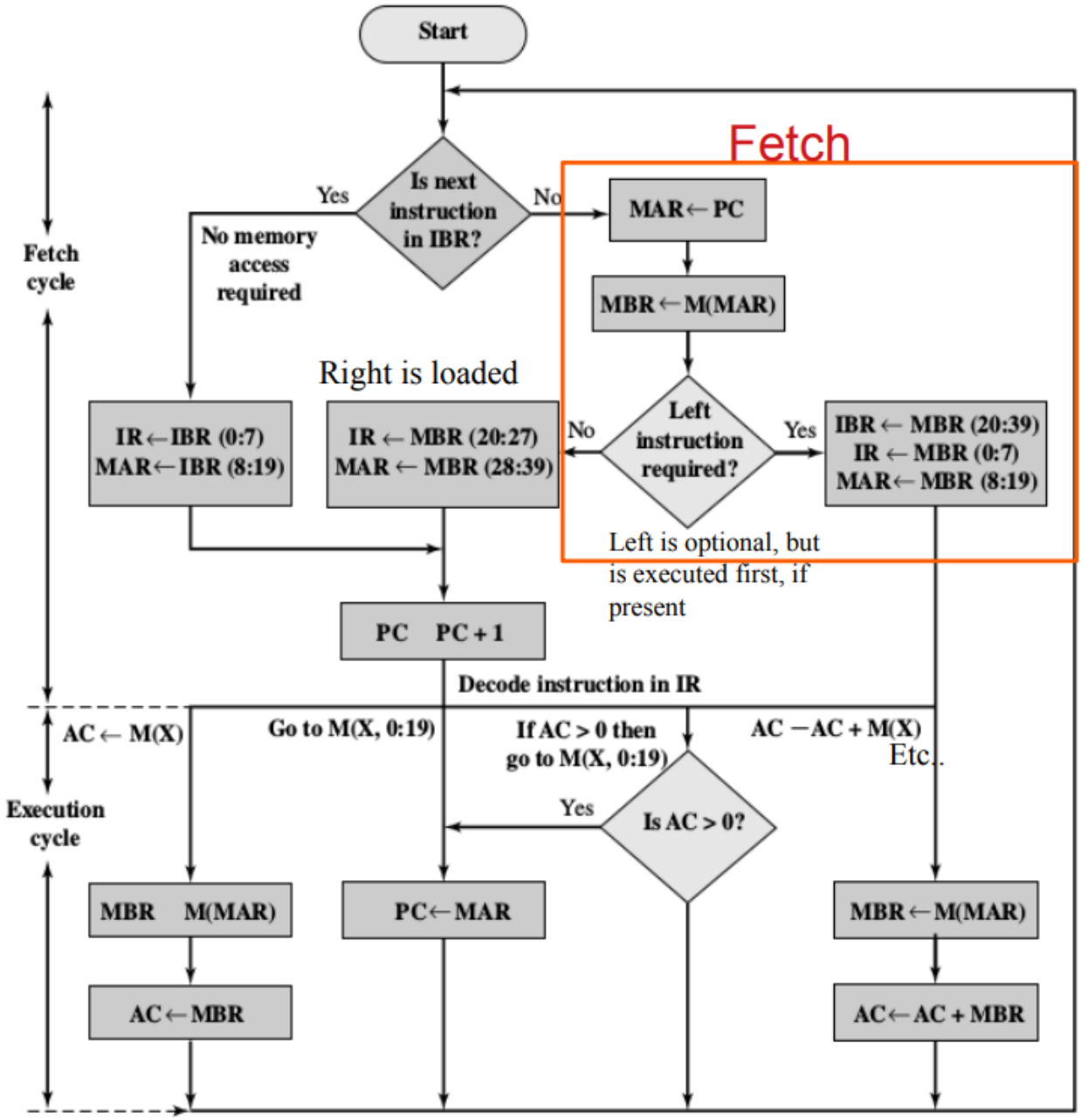

Figure 1: IAS Flow Chart

Matrix Multiplier:

The matrix multiplier program takes the entries of two 2x2 matrices and returns the matrix product of both the matrices. Obviously the last print statement is wrong but its really not a big of a deal in the real workings of the program and hence is written in a fashion that is more readable rather than implementation wise. (if you are really stingy about it then we could print each element separated by spaces)

Original C++ Code :

```cpp
#include <bits/stdc++.h>
using namespace std;

int main(){
    int x1, x2, x3, x4, y1, y2, y3, y4;
    int p, q, r, s;
    int a[2][2] = {{x1, x2},
                   {y1, y2}};
    int b[2][2] = {{x3, x4},
                   {y3, y4}};

    p = (x1 * x3) + (x2 * y3);
    q = (x1 * x4) + (x2 * y4);
    r = (y1 * x3) + (y2 * y3);
    s = (y1 * x4) + (y2 * y4);

    cout << {{p, q}, {r, s}} << endl;
}
```

Assembly Code:

Equivalent to the above C++ code we will write a assembly code in the ISA format. Our instructions start from line 1 and the memory starts from line 22. Here we have to set PC=1 in our processor code.

Assembly Code:

```
LOAD M(22) ; MUL M(26)
LOAD MQ ; NOP
STOR M(31) ; LOAD M(23)
MUL M(28) ; LOAD MQ
ADD M(31) ; STOR M(31)
LOAD M(22) ; MUL M(27)
LOAD MQ ; NOP
STOR M(32) ; LOAD M(23)
MUL M(29) ; LOAD MQ
ADD M(32) ; STOR M(32)
LOAD M(24) ; MUL M(26)
LOAD MQ ; NOP
STOR M(33) ; LOAD M(25)
MUL M(28) ; LOAD MQ
ADD M(33) ; STOR M(33)
LOAD M(24) ; MUL M(27)
LOAD MQ ; NOP
STOR M(34) ; LOAD M(25)
MUL M(29) ; LOAD MQ
ADD M(34) ; STOR M(34)
HALT ; NOP
1
0
0
1
2
0
0
2

0
0
0
0
```

Factorial:

This program calculates the factorial of a given number. In this example the number takes is 5. We are also using another variable b which is initially set to 5 and we decrement it's value with every iteration of the while loop. At the end the output(even though not displayed) is 120.

Original C++ Code :

```
#include <bits/stdc++.h>
using namespace std;

int main(){
   int a = 1;
   int b = 5;
   while(b > 0){
      a *= b;
      b--;
   }
   return 0;
}
```

Assembly Code:

Equivalent to the above C++ code we will write a assembly code in the ISA format. Our instructions start from line 1 and the memory starts from line 22. Here we have to set PC=1 in our processor code.

Assembly Code:

```
LOAD M(10) ; MUL M(11)
LOAD MQ ; STOR M(10)
LOAD M(11) ; SUB M(12)
STOR M(11) ; LOAD M(11)
JUMP++ M(1,0:19) ; STOR M(11)
HALT ; NOP




1
5
1
```

BubbleSort:

It sorts the array using the BubbleSort technique, its really complicated and gives a good insight int the constraints and bottle necks of the IAS ISA. The many functions and arguments used here are not implemented in our assembly program. Like for example our assembly program does not account for the extra func parameter.

Original C++ Code :

```cpp
#include <bits/stdc++.h>
using namespace std;
typedef long long int ll;

bool greaterThan(ll a, ll b)
{
    return (a > b);
}

void bubbleSort(vector<ll> &arr, function<bool(ll, ll)> func)
{
    bool swapped = true;
    ll temp;

    while (swapped)
    {
        swapped = false;
        for (ll i = 1; i < arr.size(); i++)
        {
            if (func(arr[i], arr[i - 1]))
            {
                temp = arr[i - 1];
                arr[i - 1] = arr[i];
                arr[i] = temp;
                swapped = true;
            }
        }
    }
}

int main(void)
{
    vector<ll> arr = {1, 2, 3, 4, 5};
    bubbleSort(arr, greaterThan);

    for (auto a : arr)
        cout << a << ' ';
    cout << endl;
}
```

Assembly Code:

Equivalent to the above C++ code we will write a assembly code in the ISA format. Our instructions start from line 1 and the memory starts from line 28. Here we have to set PC=1 in our processor code.

Assembly Code:

```
LOAD M(35) ; SUB M(29)                          // initializing N-2
SUB M(29) ; STOR M(36)
LOAD M(28); SUB M(29)                           // swap = -1
STOR M(31) ; LOAD M(32)                         // i = starting loc of array
STOR M(33) ; LOAD M(33)                         // i = 0+startOfArray
ADD M(29) ; STOR M(34)                          // storing i+1
LOAD M(34) ; STOR M(11,8:19)                    // address modifying for i+1
STOR M(13,8:19) ; STOR M(14,28:39)              // incase two STORS are required
LOAD M(33) ; STOR M(11,28:39)                   // address modifying for i
STOR M(14,8:19) ; STOR M(15,28:39)              // incase two STORS are required
LOAD M(132) ; SUB M(131)                        // first x is i+1 second is i
JUMP+ M(17,20:39) ; NOP                         // skip
LOAD M(132) ; STOR M(30)                        // swap starts here
LOAD M(131) ; STOR M(132)                       // swap goes on here
LOAD M(30) ; STOR M(131)                        // swap ends
LOAD M(31) ; ADD M(29)
STOR M(31) ; LOAD M(33)                         // incrementing swap
ADD M(29) ; STOR M(33)                          // incrementing i
LOAD M(36) ; SUB M(33)                          // checking if it is less than N-2
JUMP+ M(5,20:39) ; LOAD M(31)                   // going back to start but after swap's while loop
JUMP+ M(3,0:19) ; HALT                          // going back to the start just before swap



// memory locations

0                                               // constant always has 0
1                                               // constant always has 1
2                                               // temporary t1
-1                                              // value of swap
37                                              // mem address from where the array starts
0                                               // value of i
0                                               // value of i+1
43                                              // value of N (ending location + 1)
42                                              // value of N-2
10                                              // values of array
9
10
9
10
11
```

Assembler:

The assembler takes assembly code written in a specific format from the "assembly code" file and converts it into machine code, which is then written to the "fileName.obj" file. This Assembler also includes "converter" for converting "binary-to-decimal" and vice-versa. The memory is stored along with the instructions, replicating the the IAS machine. The shown code below is only a part of the code and now the entire code.

Assembler Script:

```
import re
import sys


class Assembler:
    def __init__(self, inputFileName: str, outputFileName: str):
        if self.__checkType((inputFileName, str), (outputFileName, str)):
            self.__inputFileName = inputFileName
            self.__outputFileName = outputFileName

    def __getOpCode(self, op: str, address="") -> str:
        """
        Returns the opcode.

        arguemnts:
            op: the instruction itself
            address: default is \"\".
            The address should be of form M(XXXX)
        """
        if (op == 'ADD' and address[0] == 'M'):
            return '00000101'

        elif (op == 'STOR' and (',' not in address)):
            return '00100001'

        elif (op == 'LOAD' and address[0:2] == 'M('):
            return '00000001'

        elif (op == 'LOAD' and address[0:2] == '-M'):
            return '00000010'

        elif (op == 'LOAD' and address[0] == '|'):
            return '00000011'
```

This code snippet is from "assembler.py".

Memory Data :

The below is the machine code (object file) we get get after running the matrix multiplier program through the assembler. The other object files are stored within the "object files" folder.

```
00000001 000000010110 00001011 000000011010
00001010 000000000000 10000000 000000000000
00100001 000000011111 00000001 000000010111
00001011 000000011100 00001010 000000000000
00000101 000000011111 00100001 000000011111
00000001 000000010110 00001011 000000011011
00001010 000000000000 10000000 000000000000
00100001 000000100000 00000001 000000010111
00001011 000000011101 00001010 000000000000
00000101 000000100000 00100001 000000100000
00000001 000000011000 00001011 000000011010
00001010 000000000000 10000000 000000000000
00100001 000000100001 00000001 000000011001
00001011 000000011100 00001010 000000000000
00000101 000000100001 00100001 000000100001
00000001 000000011000 00001011 000000011011
00001010 000000000000 10000000 000000000000
00100001 000000100010 00000001 000000011001
00001011 000000011101 00001010 000000000000
00000101 000000100010 00100001 000000100010
11111111 000000000000 10000000 000000000000
00000000000000000000000000000000000000000001
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000001
00000000000000000000000000000000000000000010
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000010

00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000
```

Processor:

Processor reads the "fileName.obj" file and accordingly changes the contents of the different registers like - MAR (Memory Address Register) , MBR (Memory Buffer Register) , IR (Instruction Register) , AC (Accumulator) and MQ (Multiplier-Quotient) through the Control-Data Path for IAS during runtime. The addresses and words are changed during runtime in the object file itself (we use the object file as the memory itself, unlike loading it in a list or sorts).

Processor Code:

```
class MainMemory:
    def __init__(self, inputFileName: str):
        self.__inputFileName = inputFileName

    def getWord(self, lineNumber: int) -> str:
        """
        Gets the word from that Memory address. If the Memeory address is a string it is considered
            to be a binary number.
        """
        if checkType((lineNumber, str)):
            lineNumber = convertToInt(lineNumber)

        linecache.checkcache(self.__inputFileName)
        line = linecache.getline(self.__inputFileName,
                                 lineNumber).rstrip("\n")

        if line == '':
            line = '0'*40

        return line

    def replaceMemoryAddr(self, lineNumber: int, memAddr: str, leftOrRight: int) -> None:
        """
        Replaces memory address with the memAddr given. If it is in integer it is converted to
            binary.
        If leftOrRight = 1, then the left address is replaced, else the right one is replaced
        """

        if leftOrRight == 0:
            leftOrRight = -1
        else:
            leftOrRight = 1
```

This code snippet is from "processor.py".

OPCode Mapping:

The Assembler has some additional added instructions which has newly assigned "OPCodes" listed below:

```
00000000 NOP
00000001 LOAD M(X)
00000010 LOAD -M(X)
00000011 LOAD |M(X)|
00000100 LOAD -|M(X)|
00000101 ADD M(X)
00000110 SUB M(X)
00000111 ADD |M(X)|
00001000 SUB |M(X)|
00001001 LOAD MQ,M(X)
00001010 LOAD MQ
00001011 MUL M(X)
00001100 DIV M(X)
00001101 JUMP M(X,0:19)
00001110 JUMP M(X,20:39)
00001111 JUMP+ M(X,0:19)
00010000 JUMP+ M(X,20:39)

00010010 STOR M(X,8:19)
00010011 STOR M(X,28:39)
00010100 LSH
00010101 RSH

00011000 JUMP++ M(X,0:19)
00011001 JUMP++ M(X,20:39)

00100001 STOR M(X)


11111111 HALT
```

Functioning of Extra Instructions :

0.1 JUMP++ M(X, 0:19) :

This function is similar to the JUMP+ function, it only works when the value in AC is positive. After jumping, it starts from executing the entire 40-bit instruction.

0.2 JUMP++ M(X, 20:39) :

This function is similar to the JUMP+ function, it only works when the value in AC is positive. After jumping, it starts executing from the right instruction.

0.3 HALT :

This marks the end of the program.

0.4 NOP :

This instruction tells the processor to skip the instruction and move on the next instruction.

Execution Results :

The below screenshots shows the contents of different registers after running all the immediate files in order.

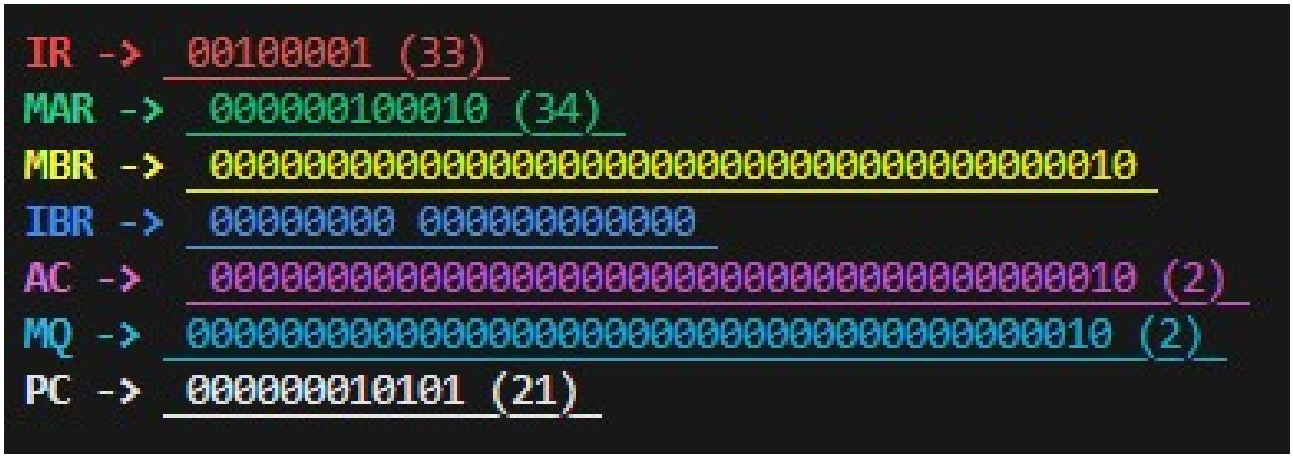

Figure 2: Result

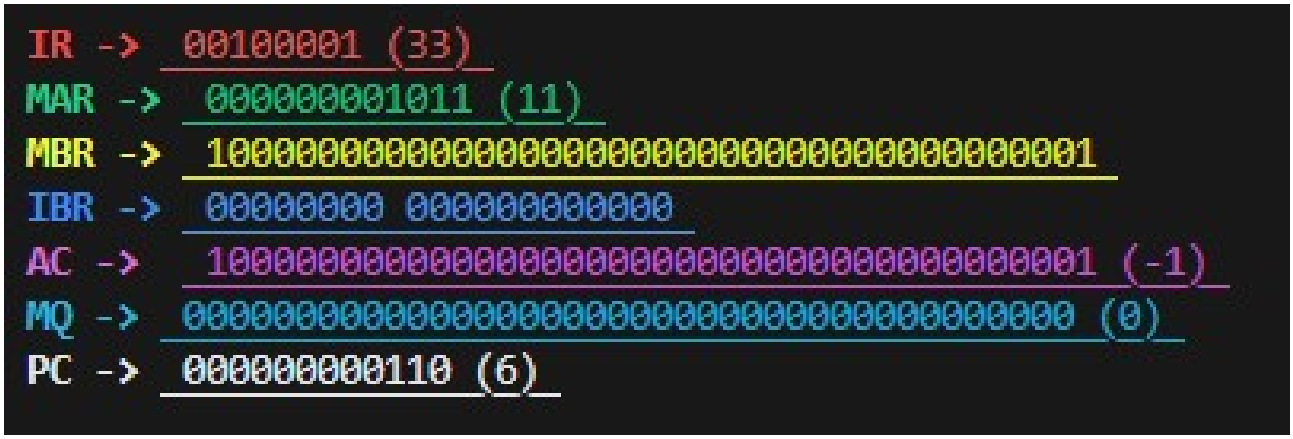

Figure 3: Result

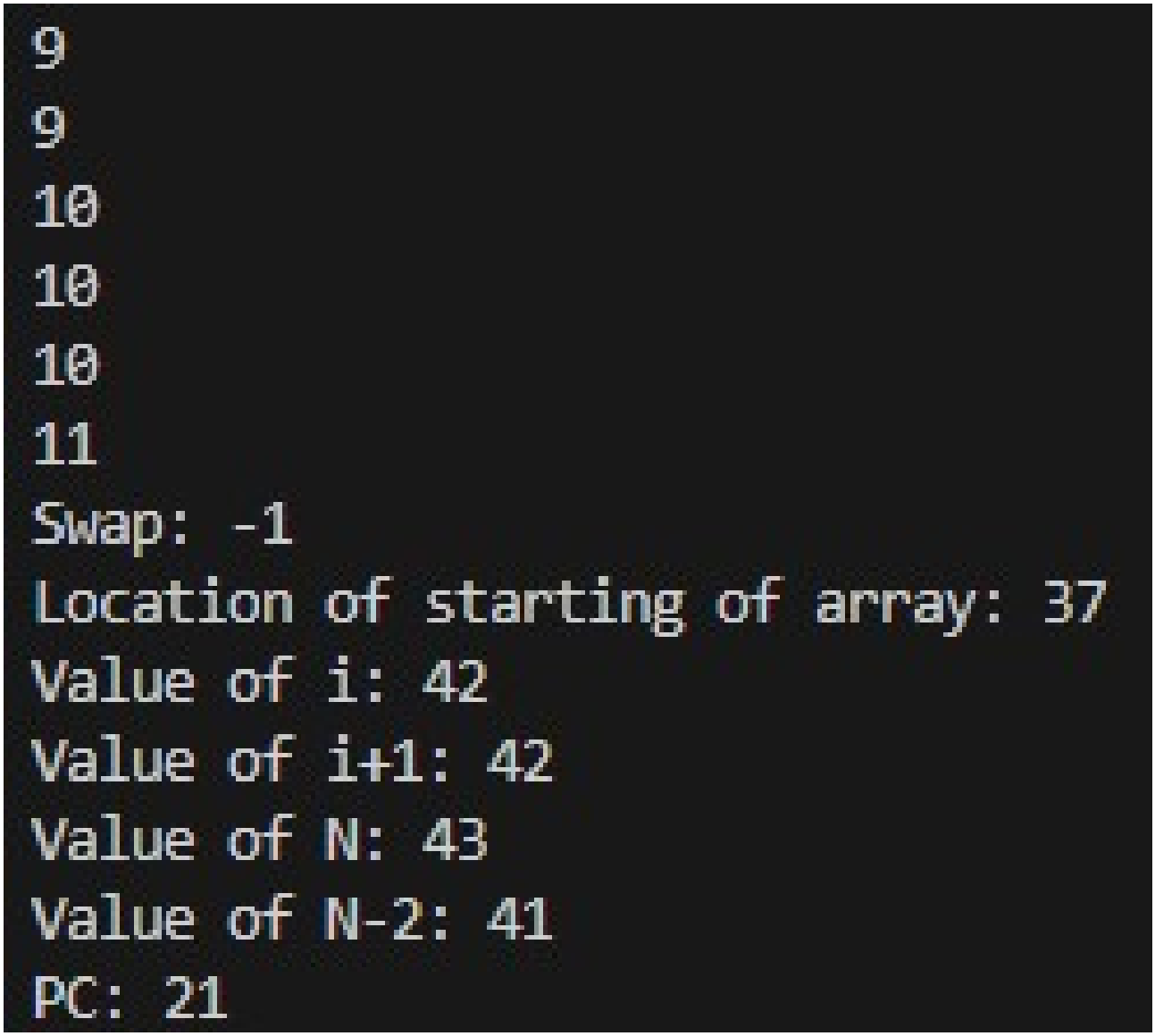

Figure 4: Result

References

[1] Wikipedia ias machine.