IAS Architecture Simulation

Course: EGC 121 - Computer Architecture

Implementation of `std::lower_bound` using IAS Assembly

Name: Pragyan Ojha

Roll No: BC2025075

International Institute of Information Technology, Bangalore

February 4, 2026

Contents

1 Introduction

The IAS machine was one of the first electronic computers built under the direction of John von Neumann. It introduced the **stored-program concept**, where instructions and data share the same memory space.

In this project, I have implemented a software simulation of the IAS architecture, including:

- A custom **Assembler** to convert assembly code to 40-bit machine code.
- A **Processor Simulator** that executes the fetch-decode-execute cycle.
- An implementation of the **Binary Search (Lower Bound)** algorithm to demonstrate the architecture's capabilities, specifically focusing on address modification.

2 Algorithm: `std::lower_bound`

Unlike a standard Binary Search which only checks for existence, the `lower_bound` function finds the *first* element in a sorted range that is greater than or equal to a target value.

2.1 Logic Flow

The algorithm maintains a search space $[L, R]$. In each iteration:

1. Calculate $MID = (L + R)/2$ using the right-shift (`RSH`) instruction.
2. Fetch $ARR[MID]$ using **self-modifying code** (replacing the address field of a dummy `LOAD` instruction).
3. Compare $ARR[MID]$ with $TARGET$:
   - If $ARR[MID] \geq TARGET$: Update potential answer ($ANS = MID$) and search left ($R = MID - 1$).
   - If $ARR[MID] < TARGET$: Search right ($L = MID + 1$).

2.2 Reference Algorithm Code in C++

```cpp
#include <iostream>
#include <vector>
using namespace std;

int main() {
    vector<int> arr = {10, 20, 24, 25, 25, 30, 40, 50, 60, 70};
    int target = 25;
    int n = arr.size();

    int L = 0;
    int R = n - 1;
    int ans = -1;


```

```
    while (L <= R) {
        int mid = (L + R) / 2;

        if (arr[mid] >= target) {
            ans = mid;
            R = mid - 1;
        }
        else {
            L = mid + 1;
        }
    }

    cout << "Found at index: " << ans << endl;
    return 0;
}
```

Listing 1: Reference Implementation of Lower Bound

3 Implementation Details

3.1 Memory Map

The memory is organized as follows to separate instructions from data:

Address	Variable	Description
0 – 99	Code	The program instructions
100 – 106	Variables	$L, R, MID, TARGET, ANS, 1$
500 – 509	Array	The sorted data array

Table 1: Memory Organization

3.2 Assembly Code

The following IAS assembly implements the `lower_bound` logic.

```
//Loop Check (L <= R)
LOAD M(101)                 //load R into AC
SUB M(100)                  //subtract L from AC (R-L)
JUMP+ M(2)                  //if AC is non-negative jump to memory
    address 2
HALT                        //halts the program if jump is not executed,
    i.e, R-L<0

//Calculate MID
LOAD M(100)                 //load L into AC
ADD M(101)                  //add R to AC (L+R)
RSH                         //right shifts AC by 1 bit, equivalent to
    dividing it by 2 so (L+R)/2
STOR M(102)                 //store the result in the MID variable

//Address Modification
LOAD M(107)                 //loads a "dummy" instruction which just says
     LOAD M(0)
ADD M(106)                  //adds the base address of array which is
    500, so instruction becomes LOAD M(500)
```

```
ADD M(102)                  //adds mid so effectively intrusction becomes
    LOAD M(500+MID)
STOR_L M(6)                 //stores only the address part of AC into
   left instruction of memory address 6

//Fetch & Compare
LOAD M(0)                   //at runtime, due to intrusction at memory
   address 5, this has been overwritten to LOAD M(500+MID)
SUB M(103)                  //subtracts our target from arr[MID]
JUMP+ M(10)                 //if result is non-negative, i.e., arr[MID]>=
   TARGET we jump to memory address 10
NOP                         //padding to cleanly start next instruction

//Go Right (L = MID + 1); we reach here if arr[MID]<TARGET
LOAD M(102)                 //load MID into AC
ADD M(105)                  //add one to AC
STOR M(100)                 //update L with the new value (MID+1)
JUMP M(0)                   //jump back to memory address 0 to restart
   the loop

//Found & Go Left (R = MID - 1); we reach here if arr[MID]>=TARGET
LOAD M(102)                 //load MID into AC
STOR M(104)                 //store AC into ANS variable (AC still
   contains MID)
SUB M(105)                  //subtract 1 from AC
STOR M(101)                 //update R with the new value (MID-1)
JUMP M(0)                   //jump back to memory address 0 to restart
   the loop

//Variables
DATA 100 0                  //L
DATA 101 9                  //R
DATA 102 0                  //MID
DATA 103 25                 //TARGET
DATA 104 -1                 //ANS
DATA 105 1                  //ONE
DATA 106 500                //BASE ADDRESS
DATA 107 4294967296         //TEMPLATE LOAD M(0)

//Sorted Array Data
DATA 500 10
DATA 501 20
DATA 502 24
DATA 503 25
DATA 504 25
DATA 505 30
DATA 506 40
DATA 507 50
DATA 508 60
DATA 509 70
```

Listing 2: Assembly Code for Lower Bound Algorithm

3.3 Assembler Code (Python)

The assembler reads the assembly_code.txt file, pairs instructions into 40-bit words and parses DATA directives to initialize variables and array elements.

```
#Standard IAS Opcodes
OPCODES = {
    'NOP': '00000000', 'HALT': '11111111',
    'LOAD': '00000001', 'ADD': '00000101', 'SUB': '00000110',
    'MUL': '00001011', 'DIV': '00001100',
    'LSH': '00010100', 'RSH': '00010101',
    'STOR': '00100001', 'STOR_L': '00010010',
    'JUMP': '00001101', 'JUMP+': '00001111'
}

def assemble(input_file, output_file):
    lines = []        #To store the final clean assembly code
    memory_data = {}  #To store variables defined via DATA directive

    #Read assembly code file
    with open(input_file, 'r') as f:
        for line in f:
            #Strip comments and whitespace
            clean = line.split('//')[0].strip().upper()
            if not clean: continue

            #Check for Assembler Directives (DATA)
            if clean.startswith("DATA"):
                parts = clean.split()
                #Format: DATA <Address> <Value>
                address = int(parts[1])
                value = int(parts[2])
                memory_data[address] = value
            else:
                lines.append(clean)

    #Convert pairs of instructions to binary pairs
    with open(output_file, 'w') as f:
        #Loop through lines 2 at a time (Left & Right Instructions)
        for i in range(0, len(lines), 2):
            pair_binary = ""

            #Process Left and Right instruction(Loop runs twice)
            for j in [i, i+1]:
                if j >= len(lines): #Padding if odd number of lines
                    pair_binary += OPCODES['NOP'] + "000000000000"
                    continue

                parts = lines[j].split()        #Splitting by space
                cmd = parts[0]

                #Get Opcode
                opcode = OPCODES.get(cmd, '00000000')

                #Get Address
                addr = 0
                if len(parts) > 1:
                    #Takes string "M(123)", slices off "M(" and ")",
   then converts to int
                    addr_str = parts[1].replace("M(", "").replace(")",
   "")
                    addr = int(addr_str)

```

```
                pair_binary += opcode + format(addr, '012b')
                #Convert the memory address to a 12 bit binary string

            f.write(pair_binary + '\n')

        #Write Data (Variables + Array) and fill each gap with 40 bit 0 string
        #Calculate where the instruction code ends
        code_lines_count = (len(lines) + 1) // 2
        current_addr = code_lines_count

        #Determine the highest memory address we need to write to
        max_addr = max(memory_data.keys()) if memory_data else code_lines_count

        while current_addr <= max_addr:
            val = memory_data.get(current_addr, 0)
            #Handle negative numbers for 40-bit binary
            if val < 0: val = (1 << 40) + val
            f.write(format(val, '040b') + '\n')
            current_addr += 1

    print(f"Done. Binary saved to {output_file}")

if __name__ == "__main__":
    assemble("assembly_code.txt", "binary.txt")
```

Listing 3: Assembler Script

3.4 IAS Simulator Code (Python)

The simulator implements the Fetch-Decode-Execute cycle, handling the 40-bit word structure (20-bit Left/Right instructions). It treats binary.txt as system memory(RAM), performing file I/O for every cycle.

```
import os

MEM_SIZE = 1000
AC = 0              #Accumulator to store result of arithmetic or load
MQ = 0              #Helper register for multiply/divide operation
MBR = 0             #Holds the current 40 bit instruction
MAR = 0             #Holds the current instruction's 12 bit memory address
IR = 0              #Holds the current instruction's 8 bit OP Code
IBR = None          #Holds the right instruction (20 bits)
PC = 0              #Holds the memory address of next instruction

#File Constants
BINARY_FILE = "binary.txt"
#Calculate line length: 40 bits + newline characters (1 for Unix, 2 for Windows)
LINE_LENGTH = 40 + len(os.linesep)

#Helper functions to safely convert python 'numbers' to bits and vice versa
def to_signed(val):
    if val & (1 << 39):
```

```
        return val - (1 << 40)
    return val
def to_unsigned(val):
    return val & 0xFFFFFFFFFF

#Helper functions for Disk-as-RAM access
def read_memory(address):
    """Reads a specific line from binary.txt"""
    if not os.path.exists(BINARY_FILE):
        return '0' * 40

    with open(BINARY_FILE, 'r') as f:
        f.seek(address * LINE_LENGTH)
        line = f.read(40)
        #Return 40 zeros if address is uninitialized/empty
        if len(line) < 40: return '0' * 40
        return line

def write_memory(address, binary_string):
    """Writes a specific line to binary.txt"""
    with open(BINARY_FILE, 'r+') as f:
        f.seek(address * LINE_LENGTH)
        f.write(binary_string)

def fetch_decode():
    global PC, MAR, MBR, IR, IBR

    #If IBR is empty, fetch new word from Memory
    if IBR is None:
        MAR = PC
        #Fetch directly from file
        MBR = int(read_memory(MAR), 2)

        #Left Instruction (Bits 0-19); Opcode: 0-7, Address: 8-19
        left_part = (MBR >> 20) & 0xFFFFF
        IR = (left_part >> 12) & 0xFF
        MAR = left_part & 0xFFF

        #Sves the right Instruction (bits 20-39) to IBR
        IBR = MBR & 0xFFFFF
    else:
        #Execute instruction in IBR
        right_part = IBR
        IR = (right_part >> 12) & 0xFF
        MAR = right_part & 0xFFF
        IBR = None
        PC += 1               #Increment PC after right instruction

def execute():
    global AC, MQ, MBR, MAR, IR, PC, IBR

    #Fetch Data for Arithmetic Ops
    if IR in [1, 5, 6, 11, 12]:
        MBR = to_signed(int(read_memory(MAR), 2))

    if IR == 0:            #NOP
        pass
    elif IR == 1:          #LOAD M(X)
```

```
78          AC = MBR
79      elif IR == 5:           #ADD M(X)
80          AC += MBR
81      elif IR == 6:           #SUB M(X)
82          AC -= MBR
83
84      elif IR == 11:          #MUL M(X) (Upper 39 bits to AC, Lower 39 to
    MQ)
85          res = AC * MBR
86          AC = res
87          MQ = 0
88      elif IR == 12:          #DIV M(X) (Quotient -> MQ, Remainder -> AC)
89          if MBR != 0:
90              MQ = AC // MBR
91              AC = AC % MBR
92          else:
93              print("Error: Division by Zero")
94
95      elif IR == 20:          #LSH (Left Shift)
96          AC = AC << 1
97      elif IR == 21:          #RSH (Right Shift)
98          AC = AC >> 1
99
100     elif IR == 33:          #STOR M(X)
101         #Write directly to file
102         write_memory(MAR, format(to_unsigned(AC), '040b'))
103
104     elif IR == 18:          #STOR_L M(X, 8:19)
105         current_word = int(read_memory(MAR), 2)      #This is the "Self
    -Modifying" instruction
106
107         #Instruction structure: Left Op(8 bits) | Left Addr(12 bits) ||
     Right Op(8 bits) | Right Addr(12 bits)
108         #We replace Bits 8-19 (The Left Address) with AC's lower 12
    bits
109
110         left_op = (current_word >> 32) & 0xFF
111         right_part = current_word & 0xFFFFF
112         new_addr = AC & 0xFFF
113
114         new_word = (left_op << 32) | (new_addr << 20) | right_part
115         #Write directly to file
116         write_memory(MAR, format(new_word, '040b'))
117
118     elif IR == 13:          #JUMP M(X, 0:19)
119         PC = MAR
120         IBR = None          #Clear IBR to force fetch from new PC
121     elif IR == 15:          #JUMP+ M(X, 0:19)
122         if AC >= 0:
123             PC = MAR
124             IBR = None
125
126     elif IR == 255:         #HALT
127         print("\nHALT ENCOUNTERED")
128         return False
129
130     return True
131
```

```python
#prints the status of each variable at every cycle
def print_trace(cycle):
    print(f"Cycle {cycle:<3} | PC: {PC:<3} | IR: {IR:<3} | MAR: {MAR:<4} | AC: {AC:<5} | MQ: {MQ:<5} | IBR: {'YES' if IBR else 'NO'}")

if __name__ == "__main__":
    print(f"{'Cycle':<9} | {'PC':<7} | {'IR':<7} | {'MAR':<8} | {'AC':<9} | {'MQ':<9} | {'IBR'}")
    print("-" * 70)

    cycle = 0
    running = True

    while running:
        cycle += 1
        fetch_decode()
        running = execute()
        print_trace(cycle)

        #Safety Break
        if cycle > 500:
            print("Terminating: Max cycles reached.")
            break

    #Dump Final Memory for Verification (Reading from file)
    print(f"\nFinal Result (ANS at 104): {int(read_memory(104), 2)}")
```

Listing 4: IAS Processor Simulator

4 Results and Verification

4.1 Generated Machine Code (binary.txt)

Below is the generated machine code by assembler.py. This is the initial state of binary.txt, i.e., before running the processor simulator(ias.py). For brevity, large blocks of empty memory (zeros) have been omitted.

```
//INSTRUCTION SPACE (Addresses 0 to 12)
0000000100000110010100000110000001100100
0000111100000000010111111111000000000000
0000000100000110010000000101000001100101
0001010100000000000000100001000001100110
0000000100000110101100000101000001101010
0000010100000110011000010010000000000110
0000000100000000000000001100000001100111
0000111100000000101000000000000000000000
0000000100000110011000000101000001101001
0010000100000110010000001101000000000000
0000000100000110011000100001000001101000
0000011000000110100100100001000001100101
0000110100000000000000000000000000000000

//[Addresses 13-99 omitted: All Zeros]

//VARIABLE SPACE (Addresses 100 to 107)
0000000000000000000000000000000000000000    // L
```

```
00000000000000000000000000000000001001    // R
00000000000000000000000000000000000000    // MID
00000000000000000000000000000000011001    // TARGET
11111111111111111111111111111111111111    // ANS
00000000000000000000000000000000000001    // ONE
00000000000000000000000000000111110100    // BASE ADDRESS
00000001000000000000000000000000000000    // TEMPLATE

//[Lines 108-499 omitted: All Zeros]

//ARRAY DATA (Addresses 500 to 509)
00000000000000000000000000000001010    // 10
00000000000000000000000000000010100    // 20
00000000000000000000000000000011000    // 24
00000000000000000000000000000011001    // 25
00000000000000000000000000000011001    // 25
00000000000000000000000000000011110    // 30
00000000000000000000000000000101000    // 40
00000000000000000000000000000110010    // 50
00000000000000000000000000000111100    // 60
00000000000000000000000000001000110    // 70
```

Listing 5: Truncated Content of binary.txt

4.2 Test Case Verification

The simulation was tested against the following standard case to verify the logic:

- **Target:** 25
- **Array:** $\{10, 20, 24, 25, 25, 30, 40, 50, 60, 70\}$
- **Expected Answer:** Index 3 (Memory Address 503) — First occurrence of 25.

I've attached the terminal screenshots showing the output. There were a total of 80 cycles but to prevent repetitiveness I've only shown cycles 1-34 and 44-80. It captures the complete flow of the simulation without feeling repetitious.

```
PS C:\Users\pragy\Downloads\Sem-2\Computer Architecture\assignments\ass 1> python -u "c:\Users\pragy\Downloads\Sem-2\Computer Arch
itecture\assignments\ass 1\assembler.py"
Done. Binary saved to binary.txt
PS C:\Users\pragy\Downloads\Sem-2\Computer Architecture\assignments\ass 1> python -u "c:\Users\pragy\Downloads\Sem-2\Computer Arch
itecture\assignments\ass 1\ias.py"
Cycle     | PC      | IR      | MAR      | AC        | MQ        | IBR
-----------------------------------------------------------------------
Cycle 1   | PC: 0   | IR: 1   | MAR: 101  | AC: 9     | MQ: 0     | IBR: YES
Cycle 2   | PC: 1   | IR: 6   | MAR: 100  | AC: 9     | MQ: 0     | IBR: NO
Cycle 3   | PC: 2   | IR: 15  | MAR: 2    | AC: 9     | MQ: 0     | IBR: NO
Cycle 4   | PC: 2   | IR: 1   | MAR: 100  | AC: 0     | MQ: 0     | IBR: YES
Cycle 5   | PC: 3   | IR: 5   | MAR: 101  | AC: 9     | MQ: 0     | IBR: NO
Cycle 6   | PC: 3   | IR: 21  | MAR: 0    | AC: 4     | MQ: 0     | IBR: YES
Cycle 7   | PC: 4   | IR: 33  | MAR: 102  | AC: 4     | MQ: 0     | IBR: NO
Cycle 8   | PC: 4   | IR: 1   | MAR: 107  | AC: 4294967296 | MQ: 0     | IBR: YES
Cycle 9   | PC: 5   | IR: 5   | MAR: 106  | AC: 4294967796 | MQ: 0     | IBR: NO
Cycle 10  | PC: 5   | IR: 5   | MAR: 102  | AC: 4294967800 | MQ: 0     | IBR: YES
Cycle 11  | PC: 6   | IR: 18  | MAR: 6    | AC: 4294967800 | MQ: 0     | IBR: NO
Cycle 12  | PC: 6   | IR: 1   | MAR: 504  | AC: 25    | MQ: 0     | IBR: YES
Cycle 13  | PC: 7   | IR: 6   | MAR: 103  | AC: 0     | MQ: 0     | IBR: NO
Cycle 14  | PC: 10  | IR: 15  | MAR: 10   | AC: 0     | MQ: 0     | IBR: NO
Cycle 15  | PC: 10  | IR: 1   | MAR: 102  | AC: 4     | MQ: 0     | IBR: YES
Cycle 16  | PC: 11  | IR: 33  | MAR: 104  | AC: 4     | MQ: 0     | IBR: NO
Cycle 17  | PC: 11  | IR: 6   | MAR: 105  | AC: 3     | MQ: 0     | IBR: YES
Cycle 18  | PC: 12  | IR: 33  | MAR: 101  | AC: 3     | MQ: 0     | IBR: NO
Cycle 19  | PC: 0   | IR: 13  | MAR: 0    | AC: 3     | MQ: 0     | IBR: NO
Cycle 20  | PC: 0   | IR: 1   | MAR: 101  | AC: 3     | MQ: 0     | IBR: YES
Cycle 21  | PC: 1   | IR: 6   | MAR: 100  | AC: 3     | MQ: 0     | IBR: NO
Cycle 22  | PC: 2   | IR: 15  | MAR: 2    | AC: 3     | MQ: 0     | IBR: NO
Cycle 23  | PC: 2   | IR: 1   | MAR: 100  | AC: 0     | MQ: 0     | IBR: YES
Cycle 24  | PC: 3   | IR: 5   | MAR: 101  | AC: 3     | MQ: 0     | IBR: NO
Cycle 25  | PC: 3   | IR: 21  | MAR: 0    | AC: 1     | MQ: 0     | IBR: YES
Cycle 26  | PC: 4   | IR: 33  | MAR: 102  | AC: 1     | MQ: 0     | IBR: NO
Cycle 27  | PC: 4   | IR: 1   | MAR: 107  | AC: 4294967296 | MQ: 0     | IBR: YES
Cycle 28  | PC: 5   | IR: 5   | MAR: 106  | AC: 4294967796 | MQ: 0     | IBR: NO
Cycle 29  | PC: 5   | IR: 5   | MAR: 102  | AC: 4294967797 | MQ: 0     | IBR: YES
Cycle 30  | PC: 6   | IR: 18  | MAR: 6    | AC: 4294967797 | MQ: 0     | IBR: NO
Cycle 31  | PC: 6   | IR: 1   | MAR: 501  | AC: 20    | MQ: 0     | IBR: YES
Cycle 32  | PC: 7   | IR: 6   | MAR: 103  | AC: -5    | MQ: 0     | IBR: NO
Cycle 33  | PC: 7   | IR: 15  | MAR: 10   | AC: -5    | MQ: 0     | IBR: NO
Cycle 34  | PC: 8   | IR: 0   | MAR: 0    | AC: -5    | MQ: 0     | IBR: NO
```

Figure 1: Terminal Output showing Register Trace. Note Cycle 8–12 where the address modification logic dynamically constructs the instruction LOAD M(504).

```
Cycle 44  | PC: 3   | IR: 21  | MAR: 0    | AC: 2     | MQ: 0     | IBR: YES
Cycle 45  | PC: 4   | IR: 33  | MAR: 102  | AC: 2     | MQ: 0     | IBR: NO
Cycle 46  | PC: 4   | IR: 1   | MAR: 107  | AC: 4294967296 | MQ: 0     | IBR: YES
Cycle 47  | PC: 5   | IR: 5   | MAR: 106  | AC: 4294967796 | MQ: 0     | IBR: NO
Cycle 48  | PC: 5   | IR: 5   | MAR: 102  | AC: 4294967798 | MQ: 0     | IBR: YES
Cycle 49  | PC: 6   | IR: 18  | MAR: 6    | AC: 4294967798 | MQ: 0     | IBR: NO
Cycle 50  | PC: 6   | IR: 1   | MAR: 502  | AC: 24    | MQ: 0     | IBR: YES
Cycle 51  | PC: 7   | IR: 6   | MAR: 103  | AC: -1    | MQ: 0     | IBR: NO
Cycle 52  | PC: 7   | IR: 15  | MAR: 10   | AC: -1    | MQ: 0     | IBR: NO
Cycle 53  | PC: 8   | IR: 0   | MAR: 0    | AC: -1    | MQ: 0     | IBR: NO
Cycle 54  | PC: 8   | IR: 1   | MAR: 102  | AC: 2     | MQ: 0     | IBR: YES
Cycle 55  | PC: 9   | IR: 5   | MAR: 105  | AC: 3     | MQ: 0     | IBR: NO
Cycle 56  | PC: 9   | IR: 33  | MAR: 100  | AC: 3     | MQ: 0     | IBR: YES
Cycle 57  | PC: 0   | IR: 13  | MAR: 0    | AC: 3     | MQ: 0     | IBR: NO
Cycle 58  | PC: 0   | IR: 1   | MAR: 101  | AC: 3     | MQ: 0     | IBR: YES
Cycle 59  | PC: 1   | IR: 6   | MAR: 100  | AC: 0     | MQ: 0     | IBR: NO
Cycle 60  | PC: 2   | IR: 15  | MAR: 2    | AC: 0     | MQ: 0     | IBR: NO
Cycle 61  | PC: 2   | IR: 1   | MAR: 100  | AC: 3     | MQ: 0     | IBR: YES
Cycle 62  | PC: 3   | IR: 5   | MAR: 101  | AC: 6     | MQ: 0     | IBR: NO
Cycle 63  | PC: 3   | IR: 21  | MAR: 0    | AC: 3     | MQ: 0     | IBR: YES
Cycle 64  | PC: 4   | IR: 33  | MAR: 102  | AC: 3     | MQ: 0     | IBR: NO
Cycle 65  | PC: 4   | IR: 1   | MAR: 107  | AC: 4294967296 | MQ: 0     | IBR: YES
Cycle 66  | PC: 5   | IR: 5   | MAR: 106  | AC: 4294967796 | MQ: 0     | IBR: NO
Cycle 67  | PC: 5   | IR: 5   | MAR: 102  | AC: 4294967799 | MQ: 0     | IBR: YES
Cycle 68  | PC: 6   | IR: 18  | MAR: 6    | AC: 4294967799 | MQ: 0     | IBR: NO
Cycle 69  | PC: 6   | IR: 1   | MAR: 503  | AC: 25    | MQ: 0     | IBR: YES
Cycle 70  | PC: 7   | IR: 6   | MAR: 103  | AC: 0     | MQ: 0     | IBR: NO
Cycle 71  | PC: 10  | IR: 15  | MAR: 10   | AC: 0     | MQ: 0     | IBR: NO
Cycle 72  | PC: 10  | IR: 1   | MAR: 102  | AC: 3     | MQ: 0     | IBR: YES
Cycle 73  | PC: 11  | IR: 33  | MAR: 104  | AC: 3     | MQ: 0     | IBR: NO
Cycle 74  | PC: 11  | IR: 6   | MAR: 105  | AC: 2     | MQ: 0     | IBR: YES
Cycle 75  | PC: 12  | IR: 33  | MAR: 101  | AC: 2     | MQ: 0     | IBR: NO
Cycle 76  | PC: 0   | IR: 13  | MAR: 0    | AC: 2     | MQ: 0     | IBR: NO
Cycle 77  | PC: 0   | IR: 1   | MAR: 101  | AC: 2     | MQ: 0     | IBR: YES
Cycle 78  | PC: 1   | IR: 6   | MAR: 100  | AC: -1    | MQ: 0     | IBR: NO
Cycle 79  | PC: 1   | IR: 15  | MAR: 2    | AC: -1    | MQ: 0     | IBR: YES

HALT ENCOUNTERED
Cycle 80  | PC: 2   | IR: 255 | MAR: 0    | AC: -1    | MQ: 0     | IBR: NO

Final Result (ANS at 104): 3
```

Figure 2: Final execution steps. The program correctly identifies the lower bound at index 3 and halts execution at Cycle 80.