

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

January 31, 2026

Contents

1	Introduction	3
2	Algorithm: std::lower_bound	3
2.1	Logic Flow	3
2.2	Reference Algorithm Code in C++	3
3	Implementation Details	4
3.1	Memory Map	4
3.2	Assembly Code	4
3.3	Assembler Code (Python)	5
3.4	IAS Simulator Code (Python)	7
4	Results and Verification	9
4.1	Generated Machine Code (binary.txt)	9
4.2	Test Case Verification	10

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++

```

1 #include <iostream>
2 #include <vector>
3 using namespace std;
4
5 int main() {
6     vector<int> arr = {10, 20, 24, 25, 25, 30, 40, 50, 60, 70};
7     int target = 25;
8     int n = arr.size();
9
10    int L = 0;
11    int R = n - 1;
12    int ans = -1;
13
14

```

```

15     while (L <= R) {
16         int mid = (L + R) / 2;
17
18         if (arr[mid] >= target) {
19             ans = mid;
20             R = mid - 1;
21         }
22         else {
23             L = mid + 1;
24         }
25     }
26
27     cout << "Found at index: " << ans << endl;
28     return 0;
29 }
```

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.

```

1 //Loop Check (L <= R)
2 LOAD M(101)           //load R into AC
3 SUB M(100)            //subtract L from AC (R-L)
4 JUMP+ M(2)             //if AC is non-negative jump to memory
   address 2
5 HALT                  //halts the program if jump is not executed,
   i.e., R-L<0
6
7 //Calculate MID
8 LOAD M(100)           //load L into AC
9 ADD M(101)             //add R to AC (L+R)
10 RSH                  //right shifts AC by 1 bit, equivalent to
   dividing it by 2 so (L+R)/2
11 STOR M(102)           //store the result in the MID variable
12
13 //Address Modification
14 LOAD M(107)           //loads a "dummy" instruction which just says
   LOAD M(0)
15 ADD M(106)             //adds the base address of array which is
   500, so instruction becomes LOAD M(500)
```

```

16 ADD M(102)           //adds mid so effectively intrusction becomes
17   LOAD M(500+MID)
18 STOR_L M(6)          //stores only the address part of AC into
19   left instruction of memory address 6
20
21 //Fetch & Compare
22 LOAD M(0)            //at runtime, due to intrusction at memory
23   address 5, this has been overwritten to LOAD M(500+MID)
24 SUB M(103)           //subtracts our target from arr[MID]
25 JUMP+ M(10)           //if result is non-negative, i.e., arr[MID] >=
26   TARGET we jump to memory address 10
27 NOP                  //padding to cleanly start next instruction
28
29 //Go Right (L = MID + 1); we reach here if arr[MID] < TARGET
30 LOAD M(102)           //load MID into AC
31 ADD M(105)           //add one to AC
32 STOR M(100)           //update L with the new value (MID+1)
33 JUMP M(0)             //jump back to memory address 0 to restart
34   the loop
35
36 //Found & Go Left (R = MID - 1); we reach here if arr[MID] >= TARGET
37 LOAD M(102)           //load MID into AC
38 STOR M(104)           //store AC into ANS variable (AC still
39   contains MID)
40 SUB M(105)           //subtract 1 from AC
41 STOR M(101)           //update R with the new value (MID-1)
42 JUMP M(0)             //jump back to memory address 0 to restart
43   the loop

```

Listing 2: Assembly Code for Lower Bound Algorithm

3.3 Assembler Code (Python)

The assembler reads the ‘assembly.txt’ file, pairs instructions into 40-bit words, and handles the data section generation.

```

1 #Standard IAS Opcodes
2 OPCODES = {
3     'NOP': '00000000', 'HALT': '11111111',
4     'LOAD': '00000001', 'ADD': '00000101', 'SUB': '00000110',
5     'MUL': '00001011', 'DIV': '00001100',
6     'LSH': '00010100', 'RSH': '00010101',
7     'STOR': '00100001', 'STOR_L': '00010010',
8     'JUMP': '00001101', 'JUMP+': '00001111'
9 }
10
11 #Variables & Array Data
12 MEMORY_DATA = {
13     100: 0, 101: 9, 102: 0, 103: 25, 104: -1, 105: 1, 106: 500,
14     #L, R, MID, TARGET, ANS, constant 1; in order
15     107: 1 << 32, #Template instruction "LOAD M(0)"
16     #Sorted Array
17     500: 10, 501: 20, 502: 24, 503: 25, 504: 25,
18     505: 30, 506: 40, 507: 50, 508: 60, 509: 70
19 }
20
21 def assemble(input_file, output_file):
22     lines = []           #To store the final clean assembly code

```

```

23
24     #Read assembly code file
25     with open(input_file, 'r') as f:
26         for line in f:
27             #Strip comments and whitespace
28             clean = line.split('//')[0].strip().upper()
29             if clean: lines.append(clean)
30
31     #Convert pairs of instructions to binary pairs
32     with open(output_file, 'w') as f:
33         #Loop through lines 2 at a time (Left & Right Instructions)
34         for i in range(0, len(lines), 2):
35             pair_binary = ""
36
37             #Process Left and Right instruction(Loop runs twice)
38             for j in [i, i+1]:
39                 if j >= len(lines): #Padding if odd number of lines
40                     pair_binary += OPCODES['NOP'] + "000000000000"
41                     continue
42
43             parts = lines[j].split()          #Splitting by space
44             cmd = parts[0]
45
46             #Get Opcode
47             opcode = OPCODES.get(cmd, '00000000')
48
49             #Get Address
50             addr = 0
51             if len(parts) > 1:
52                 #Takes string "M(123)", slices off "M(" and ")",
53                 #then converts to int
54                 addr_str = parts[1].replace("M(", "").replace(")", "")
55                 addr = int(addr_str)
56
57             pair_binary += opcode + format(addr, '012b')
58             #Convert the memory address to a 12 bit binary string
59
60             f.write(pair_binary + '\n')
61
62         #Write Data (Variables + Array) and fill each gap with 40 bit 0
63         #string
64         current_addr = (len(lines) + 1) // 2           #Calculate how
65         many code lines we wrote
66         max_addr = max(MEMORY_DATA.keys())            #Last address
67         of our memory
68
68         while current_addr <= max_addr:
69             val = MEMORY_DATA.get(current_addr, 0)
70             #Handle negative numbers for 40-bit binary
71             if val < 0: val = (1 << 40) + val
72             f.write(format(val, '040b') + '\n')
73             current_addr += 1
74
75         print(f"Done. Binary saved to {output_file}")
76
77 if __name__ == "__main__":

```

```
75 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).

```

1 MEM_SIZE = 1000
2 AC = 0                      #Accumulator to store result of arithmetic or load
3 MQ = 0                       #Helper register for multiply/divide operation
4 MBR = 0                      #Holds the current 40 bit instruction
5 MAR = 0                      #Holds the current instruction's 12 bit memory
     address
6 IR = 0                       #Holds the current instruction's 8 bit OP Code
7 IBR = None                    #Holds the right instruction (20 bits)
8 PC = 0                       #Holds the memory address of next instruction
9 MEMORY = ['0' * 40] * MEM_SIZE
#Memory(RAM): list of 1000 string, each of 40 character(0 or 1)
11
12 #Helper functions to safely convert python 'numbers' to bits and vice versa
13 def to_signed(val):
14     if val & (1 << 39):
15         return val - (1 << 40)
16     return val
17 def to_unsigned(val):
18     return val & 0xFFFFFFFFFFF
19
20 def fetch_decode():
21     global PC, MAR, MBR, IR, IBR
22
23     #If IBR is empty, fetch new word from Memory
24     if IBR is None:
25         MAR = PC
26         MBR = int(MEMORY[MAR], 2)
27
28         #Left Instruction (Bits 0-19); Opcode: 0-7, Address: 8-19
29         left_part = (MBR >> 20) & 0xFFFF
30         IR = (left_part >> 12) & 0xFF
31         MAR = left_part & 0xFFFF
32
33         #Sves the right Instruction (bits 20-39) to IBR
34         IBR = MBR & 0xFFFF
35     else:
36         #Execute instruction in IBR
37         right_part = IBR
38         IR = (right_part >> 12) & 0xFF
39         MAR = right_part & 0xFFFF
40         IBR = None
41         PC += 1                  #Increment PC after right instruction
42
43 def execute():
44     global AC, MQ, MBR, MAR, IR, MEMORY, PC, IBR
45
46     #Fetch Data for Arithmetic Ops
47     if IR in [1, 5, 6, 11, 12]:

```

```

48     MBR = to_signed(int(MEMORY[MAR], 2))
49
50     if IR == 0:           #NOP
51         pass
52     elif IR == 1:          #LOAD M(X)
53         AC = MBR
54     elif IR == 5:          #ADD M(X)
55         AC += MBR
56     elif IR == 6:          #SUB M(X)
57         AC -= MBR
58
59     elif IR == 11:          #MUL M(X) (Upper 39 bits to AC, Lower 39 to
      MQ)
60         res = AC * MBR
61         AC = res
62         MQ = 0
63     elif IR == 12:          #DIV M(X) (Quotient -> MQ, Remainder -> AC)
64         if MBR != 0:
65             MQ = AC // MBR
66             AC = AC % MBR
67         else:
68             print("Error: Division by Zero")
69
70     elif IR == 20:          #LSH (Left Shift)
71         AC = AC << 1
72     elif IR == 21:          #RSH (Right Shift)
73         AC = AC >> 1
74
75     elif IR == 33:          #STOR M(X)
76         MEMORY[MAR] = format(to_unsigned(AC), '040b')
77     elif IR == 18:          #STOR_L M(X, 8:19)
78         current_word = int(MEMORY[MAR], 2)           #This is the "Self-
      Modifying" instruction
79
80         #Instruction structure: Left Op(8 bits) | Left Addr(12 bits) ||
      Right Op(8 bits) | Right Addr(12 bits)
81         #We replace Bits 8-19 (The Left Address) with AC's lower 12
      bits
82
83         left_op = (current_word >> 32) & 0xFF
84         right_part = current_word & 0xFFFF
85         new_addr = AC & 0xFFFF
86
87         new_word = (left_op << 32) | (new_addr << 20) | right_part
88         MEMORY[MAR] = format(new_word, '040b')
89
90     elif IR == 13:          #JUMP M(X, 0:19)
91         PC = MAR
92         IBR = None           #Clear IBR to force fetch from new PC
93     elif IR == 15:          #JUMP+ M(X, 0:19)
94         if AC >= 0:
95             PC = MAR
96             IBR = None
97
98     elif IR == 255:          #HALT
99         print("\nHALT ENCOUNTERED")
100        return False
101

```

```
102     return True
103
104 #loads the binary file generated by assembler.py
105 def load_binary(filename):
106     with open(filename, 'r') as f:
107         lines = f.readlines()
108         for i, line in enumerate(lines):
109             if i < MEM_SIZE:
110                 MEMORY[i] = line.strip()
111
112 #prints the status of each variable at every cycle
113 def print_trace(cycle):
114     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'}")
115
116 if __name__ == "__main__":
117     load_binary("binary.txt")
118     print(f'{Cycle:<9} | {PC:<7} | {IR:<7} | {MAR:<8} | {AC:<9} | {MQ:<9} | {IBR}')
119     print("-" * 70)
120
121     cycle = 0
122     running = True
123
124     while running:
125         cycle += 1
126         fetch_decode()
127         running = execute()
128         print_trace(cycle)
129
130         #Safety Break
131         if cycle > 500:
132             print("Terminating: Max cycles reached.")
133             break
134
135 #Dump Final Memory for Verification
136 print(f"\nFinal Result (ANS at 104): fint(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. For brevity, large blocks of empty memory (zeros) have been omitted.

```
1 //INSTRUCTION SPACE (Lines 0 to 6)
2 0000000100000110010100000110000001100100
3 000011110000000000010111111000000000000
4 0000000100000110010000000101000001100101
5 000101010000000000000100001000001100110
6 0000000100000110101100000101000001101010
7 0000010100000110011000010010000000000110
8 00000001000000000000000000000110000001100111
9
10 // [Lines 7-99 omitted: All Zeros]
```

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 Architecture\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 Architecture\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.