IAS Architecture Simulation

Course Name: EG 212, Computer Architecture - Processor Design

Prakrititz Borah IMT2023547
Unnath Chittimalla IMT2023620
Chaitya Shah IMT2023055

26 January 2024

Abstract

Our Python-based IAS simulator, inspired by John von Neumann's Institute for Advanced Study architecture, allows users to write assembly code, processed by an assembler and compiler for execution on the simulated IAS processor. As a practical demonstration, we've integrated an array sorting program. Users can input unsorted arrays through the compiler to the memory and by running the assembly code, and the simulator showcases the sorting process, illustrating the practical application of the IAS architecture in solving computational problems. This addition enhances the educational experience, providing a hands-on example of algorithmic implementation within the simulated environment

Introduction

The IAS simulator, constructed in Python, meticulously adheres to the venerable Institute for Advanced Study (IAS) architecture conceptualized by John Von Neumann. Following the principles of the Von Neumann architecture, the simulator intricately integrates a unified memory structure, central processing unit (CPU), control unit, and arithmetic and logic unit (ALU). Building upon the classic design, our simulator introduces an innovative enhancement by incorporating two distinct memory units within the ALU, enhancing the system's capability to perform a broader spectrum of operations.

The operational workflow of the simulator is elegantly depicted in the provided flow chart. Users interact with the system by scripting assembly code, which undergoes a two-step process. Firstly, an assembler decodes the assembly code into machine code, transforming human-readable instructions into a format executable by the simulated IAS processor. Subsequently, a compiler translates the machine code into instructions understood by the processor. This emulation closely mirrors the sequence occurring in contemporary computing systems.

The simulated IAS processor, enriched by the dual-memory ALU, interprets and processes instructions, seamlessly simulating data flow and operations as dictated by the user's assembly code. The introduction of dual memory within the ALU expands the simulator's operational capabilities, allowing for a more diverse range of computations.

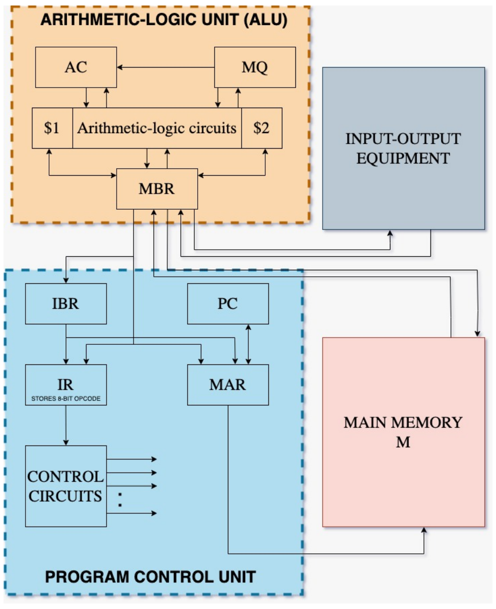


Figure 1: IAS Processor

In this project, a notable demonstration is the integration of a sorting algorithm, showcasing the practical application of the enhanced IAS architecture. This project not only pays homage to the historical roots of computing but also provides users with a hands-on experience, unraveling the inner workings of the IAS architecture and its enduring influence on the field of computer science.

Processor Implementation

The IAS Computer employs a modular design with distinct classes representing various components. The central processing unit (CPU) orchestrates the execution of instructions through the interaction of key components such as the Accumulator (AC), Arithmetic Logic Unit (ALU), Memory Address Register (MAR), Memory Buffer Register (MBR), and others.

Control Unit (CTRL)

The Control Unit manages the execution flow based on the opcode of the current instruction. The `execute()` method decodes the opcode and triggers the corresponding operation. Here's an excerpt from the `CTRL` class:

```python
class CTRL:
    def execute(self, opcode):
        # ... other cases ...
        case int(0b00000001):
            print('Calling Add')
            self.add()
        case int(0b00000010):
            print('Calling Sub')
            self.sub()
        case int(0b00000011):
            print('Calling Store')
            self.store()
        # ... other cases ...

    def add(self):
        self.MBR.update(instruction=None,
                        data=self.MEM.fetch_data(self.MAR.address))
        self.ALU.add(self.AC, self.MBR)
    # ... other functions ...
```

Code Fig: 1: CTRL class Implementation

Arithmetic Logic Unit (ALU)

The ALU handles arithmetic operations, including addition, subtraction, loading, and storing. Additionally, it supports operations related to two auxiliary registers, $1 and 2$. Here's a snippet from the `ALU` class:

```python
class ALU:
    def add(self, ac, mbr):
        ac.val += mbr.data
        print('Added', mbr.data,
              'to the accumulator. Now, the accumulator has value', ac.val,
              'in it.')

    def sub(self, ac, mbr):
        ac.val -= mbr.data
        print('Subtracted', mbr.data,
              'from the accumulator. Now, the accumulator has value', ac.val,
              'in it')

    def load(self, ac, mbr):
        ac.val = mbr.data
        print('Updated AC value to', mbr.data, '.')

    def store(self, ac, mbr):
        mbr.update(instruction=None, data=ac.val)
        print('Loaded AC value to MBR.')
    # ... other methods ...
```

Code Fig: 2: ALU Class Implementation

Instruction Buffer Register (IBR) and Instruction Register (IR)

The IBR class stores the right instruction (ri) and right address (ra) obtained from the Memory Buffer Register (MBR). The IR class holds the opcode of the instruction. These are crucial components for the execution flow. Here's a snippet from the IBR and IR classes:

```python
class IBR:
    def update(self, instruction):
        self.instruction = instruction
        self.ri, self.ra = instruction[0], instruction[1]
        print('Updated IBR to', self.instruction)

class IR:
    def update(self, opcode):
        self.opcode = opcode
        print('Updated IR to', self.opcode)
```

Code Fig: 3: IBR and IR Classes Implentation

These snippets provide an overview of how the processor classes are structured and how they interact to execute instructions within the IAS Computer. The complete implementation involves other classes such as AC, MAR, MBR, MEM, and PC, which collectively contribute to the functionality of the IAS Computer.

IAS Computer Execution Loop

The core execution loop of the IAS Computer is encapsulated within the start() method of the IASComputer class. This method initiates the execution of instructions by fetching them from memory, decoding, and executing them until the Program Counter (PC) reaches the specified number of lines. Here's a snippet from the IASComputer class:

```python
class IASComputer:
     # ... other methods ...

    def start(self, lines):
        print('lines = ', lines)
        while (self.PC.val <= lines):
            print('We are in PC = ', self.PC.val)
            print('IBR has', self.IBR.instruction)
            if (self.IBR.ri == 0b00000000):
                # ... other cases ...
            else:
                self.IR.update(self.IBR.ri)
                self.MAR.update(self.IBR.ra)
                self.IBR.clear()
                self.PC.update()
                self.CTRL.execute(self.IR.opcode)
        print(self.MEM.memory)
```

Code Fig: 4: IAS Computer Execution Loop

In this loop, the IAS Computer fetches instructions, decodes them, and executes the corresponding operations based on the opcode. The loop continues until the Program Counter (PC) reaches the specified number of lines.

Execution Loop Visualization

To provide a visual representation of the execution loop, a flow chart has been created. Figure 2 illustrates the sequence of operations performed during each iteration of the loop.

The flowchart visually guides the reader through the steps involved in fetching, decoding, and executing instructions within the IAS Computer execution loop.

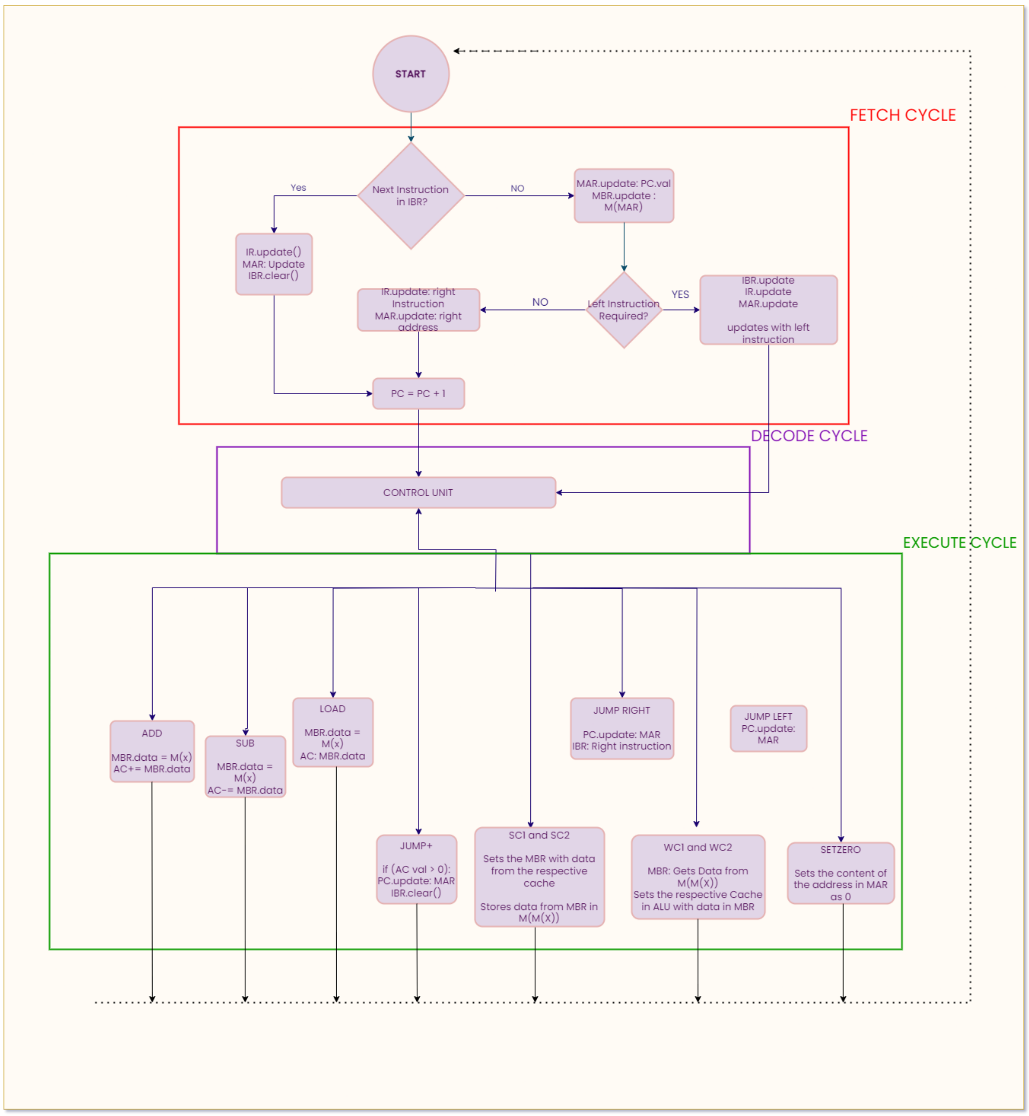

START
FETCH CYCLE
Next Instruction in IBR?
Yes
NO
MAR.update: PC.val MBR.update : M(MAR)
IR.update() MAR: Update IBR.clear()
IR.update: right Instruction MAR.update: right address
NO
Left Instruction Required?
YES
IBR.update IR.update MAR.update updates with left instruction
PC = PC + 1
DECODE CYCLE
CONTROL UNIT
EXECUTE CYCLE
ADD MBR.data = M(x) AC+= MBR.data
SUB MBR.data = M(x) AC-= MBR.data
LOAD MBR.data = M(x) AC: MBR.data
JUMP RIGHT PC.update: MAR IBR: Right instruction
JUMP LEFT PC.update: MAR
JUMP+ if (AC val > 0): PC.update: MAR IBR.clear()
SC1 and SC2 Sets the MBR with data from the respective cache Stores data from MBR in M(M(x))
WC1 and WC2 MBR: Gets Data from M(M(x)) Sets the respective Cache in ALU with data in MBR
SETZERO Sets the content of the address in MAR as 0

Figure 2: Execution Loop Flowchart

Assembler

The assembler plays a crucial role in translating human-readable assembly code into machine code understandable by the IAS computer. The Python script `decode_code` accomplishes this task by reading an input text file, decoding each line, and mapping the instructions to their respective opcodes.

Assembler Script

The following Python script exemplifies the functionality of the assembler. The `decode_code` function, when given the path to an assembly code file as input, generates the corresponding machine code. The decoded machine code is then both printed to the console and saved in a file named `Machine.out`.

```python
import sys

def decode_code(code_path):
    # ... (same as your provided script)

code_path = sys.argv[1]
decoded_code = decode_code(code_path)
```

```

# Print the decoded machine code to the console
for line in decoded_code:
    print(line)

# Save the decoded machine code to a file named Machine.out
with open("Machine.out", "w") as txt_file:
    for line in decoded_code:
        if line == decoded_code[-1]:
            txt_file.write(line)
        else:
            txt_file.write(line + "\n")

print('Machine code dumped into Machine.out')
```

Code Fig: 5: Assembler Script

To run the script in the shell, use the following command:

```
$ python3 assembler.py required_assemblycode.asm
```

Replace `required_assemblycode.asm` with the path to the assembly code file you want to process. This command assumes that you are running the script using Python 3. Adjust the command accordingly if you are using a different version of Python.

Opcode Mapping

The assembler uses a mapping table to associate assembly instructions with their corresponding opcodes. The following table provides an overview of various opcodes and their associated functions:

Opcode	Function
00000001	ADD
00000010	SUB
00000011	STORE
00000100	LOAD
00000101	JUMP_RIGHT
00000110	JUMP_LEFT
00000111	JUMP+
00010000	SETZERO
00010001	WC1
00010010	WC2
00100000	SC1
00100001	SC2
10000000	NOP
01000000	CHECKC1C2

This mapping table facilitates the translation process, allowing the assembler to convert assembly code into its machine code representation.

Compiler and Memory Initialization

The compiler, along with the memory initialization, is a crucial step in preparing the IAS computer for code execution. The provided Python script utilizes the `write_code` function to load machine code from `Machine.out` into memory. Additionally, specific memory locations are initialized to set the stage for program execution.

Memory Initialization

The memory initialization involves setting specific memory locations to predefined values. Here's a snippet from the Python script:

```python
from new_processor import *

code_location = 'Machine.out'
memory_location = 0b00000001

# ... (existing code)

# Initializing memory locations for array elements and variables
computer.MEM.write(200, 3)  # Array elements
computer.MEM.write(201, 1)
computer.MEM.write(202, 2)
computer.MEM.write(203, 7)
computer.MEM.write(204, 9)
computer.MEM.write(205, 4)

# Initializing memory locations for array processing
computer.MEM.write(250, 200)  # Array starting index
computer.MEM.write(251, 0)    # i
computer.MEM.write(252, 0)    # j
computer.MEM.write(253, 4)    # N-2
computer.MEM.write(254, 1)    # 1
computer.MEM.write(255, 1)    # temp2

print('After loading code and memory initialization, ', computer.MEM.memory)
```

Code Fig: 6: Memory Initialization

This snippet sets specific memory locations to initial values, including array elements and variables used in the program.

Code Loading and Execution

The `write_code` function reads machine code from `Machine.out` and loads it into memory. The IAS computer is then initialized, and the program is executed using the `start()` method:

```python
# ... (existing code)

# Loading machine code into memory
lines = write_code(code_location, memory_location, computer.MEM)

# Displaying the initial state of the memory
print('After loading code and memory initialization, ', computer.MEM.memory)

# Starting the IAS computer execution
computer.start(lines)
```

Code Fig: 7: Code Loading and Execution

This completes the compiler and memory initialization, preparing the IAS computer for the execution of the loaded program.

To compile and run the program in the shell, use the following command:

```
$ python3 compile.py Machine.out
```

Replace `compile.py` with the actual name of your Python script for compilation, and `Machine.out` with the file output from the assembler. This command assumes that you are running the script using Python 3. Adjust the command accordingly if you are using a different version of Python.

Bubble Sort in IAS Assembly

Original C Code

```c
#include <stdio.h>

void bubble(int *a, int n) {
  int i = 0, j;
  while (i < n - 1) {
    j = 0;
    while (j < n - i - 1) {
      if (a[j] > a[j + 1]) {
        // SWAP
        a[j] = a[j] + a[j + 1];
        a[j + 1] = a[j] - a[j + 1];
        a[j] = a[j] - a[j + 1];
      }
      j++;
    }
    i++;
  }
}
int main() {
  int arr[] = {3, 1, 2, 7, 9, 4};
  int n = sizeof(arr) / sizeof(arr[0]);
  bubble(arr, 6);

  for (int i = 0; i < n; i++) {
    printf("%d ", arr[i]);
  }
  printf("\n");
  return 0;
}
```

Code Fig: 8: Original Bubble Sort in C

To implement a simple C bubble sort code in IAS assembly, we'll leverage the provided initialization and create an assembly code that corresponds to the C code. The following assembly code demonstrates the translation:

```
SETZERO M(251) NOP #Initialize i to zero
SETZERO M(252) NOP #Initialize j to zero
----LOOP_COMPARE----
  LOAD M(250) ADD M(252)  # Calculate address of a[j]
  STORE M(255) WC1 M(255)  # Store a[j] in $1
  ADD M(254) STORE M(255)  # Calculate address of a[j+1]
  WC2 M(255) CHECKC1C2  # Store a[j+1] in $2 and Compare a[j] and a[j+1]
  JUMP+ END_COMPARE  # Jump to the end if a[j] > a[j+1]
  SC1 M(255) #Swap a[j] and a[j+1]
  LOAD M(255) SUB M(254)
  STORE M(255) SC2 M(255)
----END_COMPARE----
  LOAD M(252) ADD M(254)  #Increment j
  STORE M(252) LOAD M(253) SUB M(251) SUB M(252);#We now have (N-2-j-i) in AC
  JUMP+ LOOP_COMPARE   #Now,the condition is equivalent to if (j < n-1-i)
  LOAD M(251) ADD M(254)  #Increment i
  STORE M(251) LOAD M(253) SUB M(251) #(N-2-i) is in AC
  JUMP+ LOOP_COMPARE  #Condition is if( i < n-1 )
  NOP  # End of the program (NOP doesn't necessarily mean end)
```

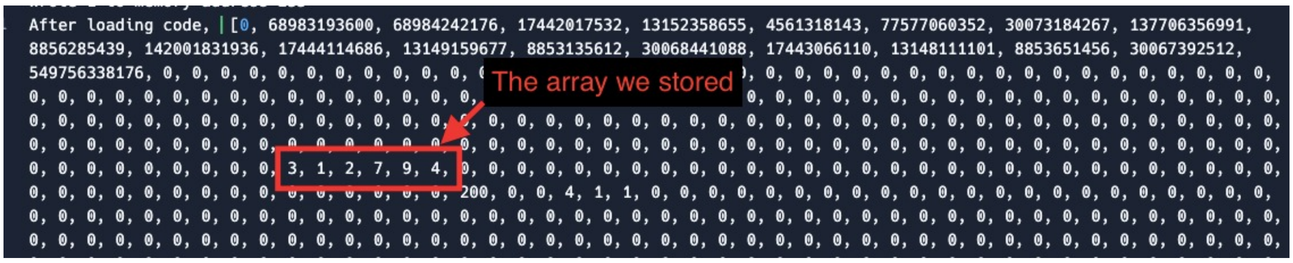
Code Fig: 9: IAS Assembly for Bubble Sort

This assembly code mimics the logic of the bubble sort algorithm. The registers $1 and 2$ are used to hold the values of ‘a[j]‘ and ‘a[j+1]‘, respectively. The ‘CHECKC1C2‘ instruction is utilized to compare these values and trigger the swap if necessary.

Note: This is a simplified example, and in a real-world scenario, additional considerations and instructions may be needed for a complete and efficient implementation. Adjustments may be necessary based on the actual memory organization and instruction set of the architecture.

Execution Results

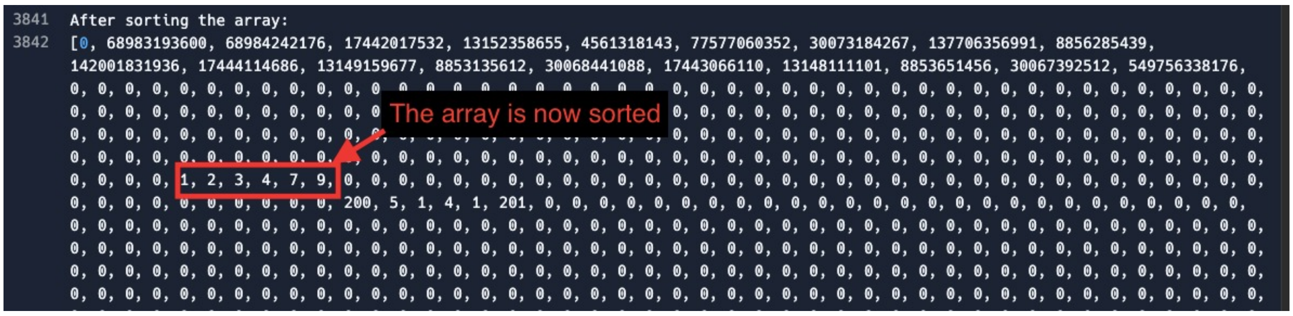
Memory Before Bubble Sort



```
After loading code, [0, 68983193600, 68984242176, 17442017532, 13152358655, 4561318143, 77577060352, 30073184267, 137706356991,
8856285439, 142001831936, 17444114686, 13149159677, 8853135612, 30068441088, 17443066110, 13148111101, 8853651456, 30067392512,
549756338176, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 3, 1, 2, 7, 9, 4, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 200, 0, 0, 4, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 3: Memory Snapshot Before Bubble Sort

Memory After Bubble Sort



```
After sorting the array:
[0, 68983193600, 68984242176, 17442017532, 13152358655, 4561318143, 77577060352, 30073184267, 137706356991, 8856285439,
142001831936, 17444114686, 13149159677, 8853135612, 30068441088, 17443066110, 13148111101, 8853651456, 30067392512, 549756338176,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 1, 2, 3, 4, 7, 9, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 200, 5, 1, 4, 1, 201, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

Figure 4: Memory Snapshot After Bubble Sort

IAS Machine Output

The detailed process of each component of the IAS machine during the bubble sort execution is documented in the output text file.

```
......... check the output.txt file to see the detailed inner execution..........
```