International Institute of Information Technology Bangalore

2026

IAS ARCHITECTURE DESIGN

Prepared by :

Arush Kumar Jain

Roll No: BC2025013

Prepared for :

Prof.Karthikeyan Vaidyanathan

Course: EGC 121 – Computer Architecture

IAS ARCHITECTURE DESIGN

Course Name: EGC 121, Computer Architecture

Introduction:

The IAS machine, developed at the Institute for Advanced Study in Princeton between 1946 and 1951, represents a seminal milestone in computer history. Often referred to as the von Neumann machine, it was constructed under the direction of John von Neumann, though its design was a collaborative effort. This machine formalized the Stored-Program Concept, where both instructions and data reside in a shared memory space,a paradigm now known as the von Neumann Architecture.

The physical implementation utilized approximately 1,000 memory locations composed of vacuum tubes. Key architectural components included:

- Central Control Unit: Responsible for fetching and decoding instructions.
- Arithmetic Logic Unit (ALU): Executed binary operations.
- Memory System: Supported random access and dynamic word modification, allowing the processor to treat instructions as data when necessary.

Assembler:

The Assembler is a Python-based utility designed to translate mnemonic assembly language into machine-executable object files. Its primary logic functions as a text-replacement engine with integrated comment stripping (handling lines prefixed with //).

To enhance the standard IAS instruction set, the following commands were implemented:

- JUMP++: A conditional branch that triggers if the Accumulator (AC) register contains a value greater than zero.
- HALT: Terminates program execution.
- NOP (No Operation): Acts as a placeholder to signify empty instruction slots or to align memory locations.

Processor:

The architectural simulation of the processor is predicated on an object-oriented paradigm, encapsulating hardware entities into discrete class structures. Rather than pursuing an excessively granular modularity, the implementation prioritizes the functional fidelity of the IAS Instruction Set Architecture (ISA). The core logic is distributed across specialized classes representing Main Memory, the Instruction Buffer Register (IBR), the Memory Address Register (MAR), and the Central Processing Unit (CPU).

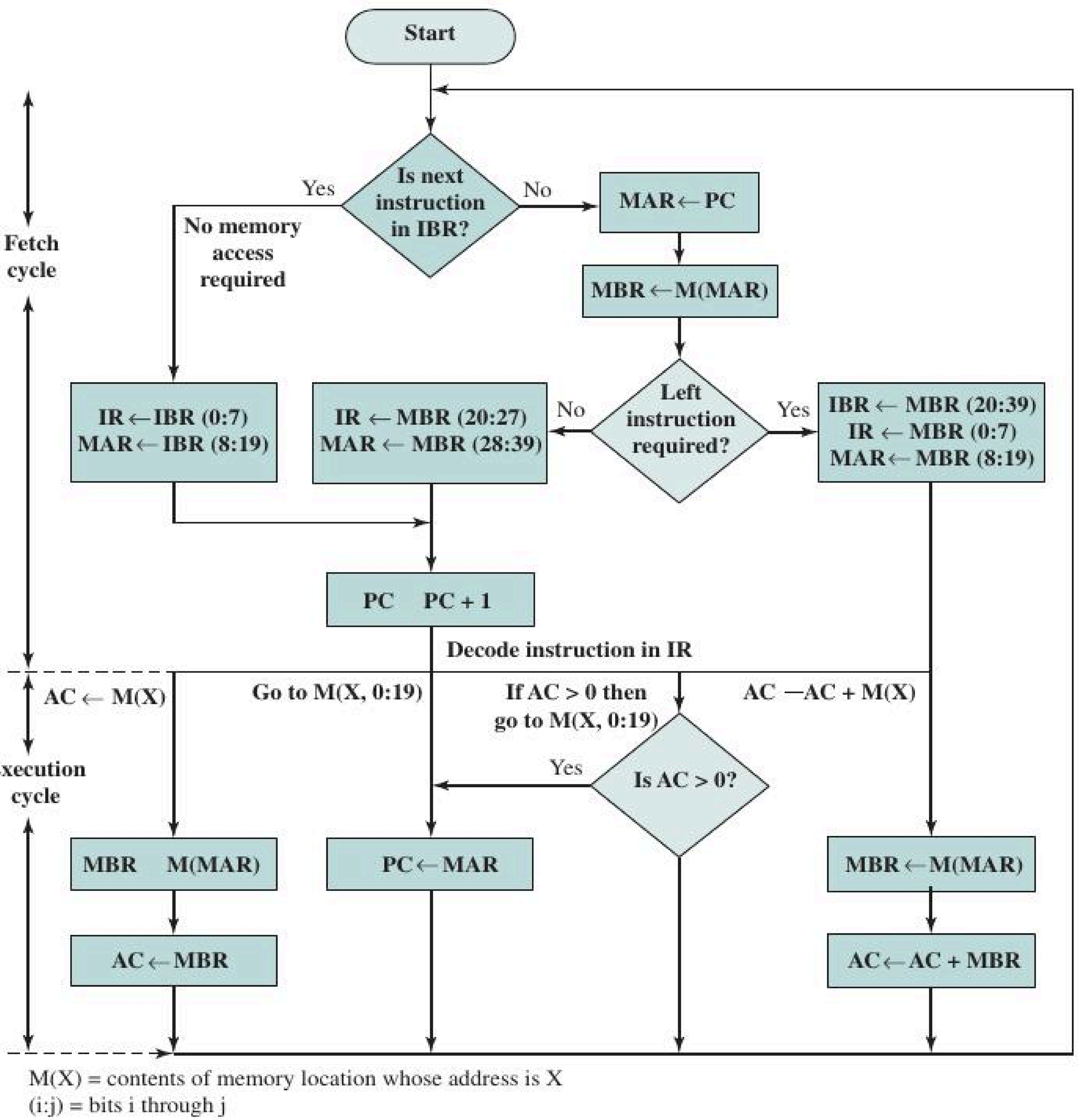

Figure 1: IAS Flow Chart

Fibonacci Sequence Generator:

The Fibonacci generator program computes sequence terms by iteratively aggregating the preceding two values stored within the system memory. The computational logic leverages the Accumulator (AC) for arithmetic summation and utilizes designated temporary memory addresses to facilitate the shifting of values, effectively simulating a control loop through the JUMP+ instruction.

While the register-level output may appear computationally dense, it is intentionally structured to provide transparency into raw data movement across the architectural bus rather than presenting a mere abstracted result. This implementation demonstrates the efficiency of iterative cycles over recursive overhead in early Von Neumann architectures. (Should a more concise representation be required, the system can be configured to output only the terminal memory states at addresses M(100) and M(101)).

Original Python Code:

```
a = 0
b = 1
count = 8
temp = 0

while count >= 0:
    temp = a + b
    a = b
    b = temp
    count = count - 1

print(a, b)
```

Assembly Code:

```
.data 100 0
.data 101 1
.data 102 10
.data 103 0
.data 105 1
LOAD M(100)
ADD M(101)
STOR M(103)
LOAD M(101)
STOR M(100)
LOAD M(103)
STOR M(101)
LOAD M(102)
SUB M(105)
STOR M(102)
JUMP++ M(0,0:19)
LOAD M(101)
HALT
```

Assembler:

The assembler takes assembly code written in a specific format from the assembly.txt file and converts it into machine-level binary instructions, which are then written to the binary.txt file. This assembler also includes a converter module for converting binary to decimal and decimal to binary, which is used during instruction encoding and data handling.Both instructions and memory are stored together, replicating the architecture of the IAS machine.
The assembly program used here generates the Fibonacci sequence, and the corresponding binary instructions are loaded into memory for execution.The code shown below represents only a part of the implementation; the complete assembler script is provided afterwards.

Assembler Script:

```
import sys
def main():
    opcodes = {
        'HALT': '11111111',
        'NOP': '10000000',
        'LOAD': '00000001',
        'LOAD-': '00000010',
        'LOAD|': '00000011',
        'LOAD-|': '00000100',
        'LOADMQ': '00001010',
        'LOADMQ_M': '00001001',
        'STOR': '00100001',
        'ADD': '00000101',
        'SUB': '00000110',
        'ADD|': '00000111',
        'SUB|': '00001000',
        'MUL': '00001011',
        'DIV': '00001100',
        'LSH': '00010100',
        'RSH': '00010101',
        'STOR_L': '00010010',
        'STOR_R': '00010011'}
    jump_ops = {
        'JUMP': {'left': '00001101', 'right': '00001110'},
        'JUMP+': {'left': '00001111', 'right': '00010000'},
        'JUMP++': {'left': '00011000', 'right': '00011001'}}
    input_file = "assembly.txt"
    output_file = "binary.txt"
    memory = []
    for i in range(1024):
        memory.append("0" * 40)
    try:
        with open(input_file, 'r') as f:
            raw_lines = f.readlines()
    except:
        print("Error: cant open file")
        return
    lines = []
    for line in raw_lines:
        line = line.split('//')[0].strip()
        if len(line) > 0:
            lines.append(line)
```

```
pc = 0
is_left = True
temp_pair = ["", ""]
for line in lines:
    if line.startswith('.data'):
        parts = line.split()
        addr = int(parts[1])
        val = int(parts[2])
        if val < 0:
            val = (1 << 40) + val
        memory[addr] = format(val, '040b')
        continue
    parts = line.split()
    op_name = parts[0]
    arg = ""
    if len(parts) > 1:
        arg = parts[1]
    addr_val = 0
    is_right_target = False

    if "M(" in arg:
        clean = arg.replace("M(", "").replace(")", "")
        if "," in clean:
            addr_str, range_str = clean.split(",")
            addr_val = int(addr_str)
            if "20:39" in range_str:
                is_right_target = True
        else:
            addr_val = int(clean)

    bin_op = "00000000"
    if op_name in opcodes:
        bin_op = opcodes[op_name]
    elif op_name in jump_ops:
        if is_right_target:
            bin_op = jump_ops[op_name]['right']
        else:
            bin_op = jump_ops[op_name]['left']
    else:
        print("Unknown opcode:", op_name)

    bin_addr = format(addr_val, '012b')
    instr = bin_op + bin_addr

    if is_left:
        temp_pair[0] = instr
        is_left = False
    else:
        temp_pair[1] = instr
        full_word = temp_pair[0] + temp_pair[1]
        memory[pc] = full_word
        pc += 1
        is_left = True
        temp_pair = ["", ""]
    if not is_left:
        full_word = temp_pair[0] + "00000000000000000000"
        memory[pc] = full_word

    with open(output_file, 'w') as out:
        for word in memory:
            out.write(word + "\n")
    print("binary file is updated now bhaiya")
main()
```

Assembler:

The following is the machine code (binary.txt) generated after running the Fibonacci sequence program through the assembler.

Binary.txt:

```
0. 00000001 000001100100 00000101 000001100101
1. 00100001 000001100111 00000001 000001100101
2. 00100001 000001100100 00000001 000001100111
3. 00100001 000001100101 00000001 000001100110
4. 00000110 000001101001 00100001 000001100110
5. 00001111 000000000000 11111111 000000000000
6. 00000000 000000000000 00000000 000000000000
7. 00000000 000000000000 00000000 000000000000
8. 00000000 000000000000 00000000 000000000000
```

↓ Addresses 9 to 99 are skipped zeros

```
100. 00000000000000000000000000000000000000
101. 00000000000000000000000000000000000001
102. 00000000000000000000000000000000001000
102. 00000000000000000000000000000000000000
104. 00000000000000000000000000000000000000
105. 00000000000000000000000000000000000001
```

↓ Addresses 106 to 1024 are skipped zeros

Processor:

The processor reads the binary.txt file and accordingly updates the contents of the architectural registers such as the Program Counter (PC), Memory Address Register (MAR), Memory Buffer Register (MBR), Instruction Register (IR), Instruction Buffer Register (IBR), Accumulator (AC), and Multiplier–Quotient (MQ) through the IAS control–data path during execution.
The memory addresses and word contents are dynamically modified within the binary.txt file itself, which is treated as the main memory of the system, rather than being loaded into auxiliary data structures during runtime.

Processor Code:

```
import sys
ac = 0
mq = 0
mbr = 0
ir = 0
mar = 0
pc = 0
ibr = 0
has_right = False
halt = False
def read_mem(addr):
    try:
        with open("binary.txt", "r") as f:
            lines = f.readlines()
            if addr < len(lines):
                return lines[addr].strip()
            else:
                return "0" * 40
    except:
        print("Error: Could not read binary.txt")
        sys.exit(1)

def write_mem(addr, val):
    try:
        with open("binary.txt", "r") as f:
            lines = f.readlines()
        while len(lines) <= addr:
            lines.append("0" * 40 + "\n")
        lines[addr] = val + "\n"
        with open("binary.txt", "w") as f:
            f.writelines(lines)
    except:
        print("Error: Could not write to binary.txt")
        sys.exit(1)

def to_int(binary_str):
    val = int(binary_str, 2)
    if binary_str[0] == '1':
        val -= (1 << 40)
    return val

def to_bin(val, bits):
    val = int(val) & ((1 << bits) - 1)
    fmt = '0' + str(bits) + 'b'
    return format(val, fmt)

def print_registers():
    print("-" * 40)
    print("PC  : " + to_bin(pc, 12))
    print("AC  : " + to_bin(ac, 40))
    print("MQ  : " + to_bin(mq, 40))
    # Shift IR to show only the 8-bit Opcode
    print("IR  : " + to_bin(ir >> 12, 8))
    print("MBR : " + to_bin(mbr, 40))
    print("MAR : " + to_bin(mar, 12))
    print("-" * 40)
```

Processor Code:

```
def fetch():
    global ir, ibr, pc, mar, mbr, has_right
    if has_right:
        ir = ibr
        has_right = False
        pc += 1
    else:
        mar = pc
        val_str = read_mem(mar)
        mbr = int(val_str, 2)
        ir = (mbr >> 20) & 0xFFFFF
        ibr = mbr & 0xFFFFF
        has_right = True

def execute():
    global ac, mq, pc, mar, mbr, ibr, has_right, halt
    op = (ir >> 12) & 0xFF
    addr = ir & 0xFFF
    mar = addr
    if op == 1:
        val = read_mem(addr)
        ac = to_int(val)
    elif op == 2:
        val = read_mem(addr)
        ac = -to_int(val)
    elif op == 3:
        val = read_mem(addr)
        ac = abs(to_int(val))
    elif op == 4:
        val = read_mem(addr)
        ac = -abs(to_int(val))
    elif op == 5:
        val = read_mem(addr)
        ac += to_int(val)
    elif op == 6:
        val = read_mem(addr)
        ac -= to_int(val)
    elif op == 7:
        val = read_mem(addr)
        ac += abs(to_int(val))
    elif op == 8:
        val = read_mem(addr)
        ac -= abs(to_int(val))
    elif op == 9:
        val = read_mem(addr)
        mq = to_int(val)
    elif op == 10:
        ac = mq
    elif op == 11:
        val = to_int(read_mem(addr))
        ac = ac * val
        mq = ac
    elif op == 12:
        val = to_int(read_mem(addr))
        if val != 0:
            mq = ac // val
            ac = ac % val
```

Processor Code:

```
    elif op == 13:
        pc = addr
        has_right = False
    elif op == 14:
        pc = addr
        mar = pc
        val_str = read_mem(mar)
        mbr = int(val_str, 2)
        ibr = mbr & 0xFFFFF
        has_right = True
    elif op == 15:
        if ac >= 0:
            pc = addr
            has_right = False
    elif op == 16:
        if ac >= 0:
            pc = addr
            mar = pc
            val_str = read_mem(mar)
            mbr = int(val_str, 2)
            ibr = mbr & 0xFFFFF
            has_right = True
    elif op == 18:
        old_val = read_mem(addr)
        new_addr_bin = to_bin(ac & 0xFFF, 12)
        new_val = old_val[:8] + new_addr_bin + old_val[20:]
        write_mem(addr, new_val)
    elif op == 19:
        old_val = read_mem(addr)
        new_addr_bin = to_bin(ac & 0xFFF, 12)
        new_val = old_val[:28] + new_addr_bin
        write_mem(addr, new_val)
    elif op == 20:
        ac = ac << 1
    elif op == 21:
        ac = ac >> 1
    elif op == 24:
        if ac > 0:
            pc = addr
            has_right = False
    elif op == 25:
        if ac > 0:
            pc = addr
            mar = pc
            val_str = read_mem(mar)
            mbr = int(val_str, 2)
            ibr = mbr & 0xFFFFF
            has_right = True
    elif op == 33:
        bin_str = to_bin(ac, 40)
        write_mem(addr, bin_str)
        mbr = ac
    elif op == 255:
        halt = True
```

Processor Code:

```
def run():
    cycles = 0
    print("Simulating IAS Machine...")
    while not halt and cycles < 2000:
        fetch()
        print("\n[Cycle " + str(cycles) + ": FETCH Completed]")
        print_registers()
        input("Press Enter to Execute...")
        execute()
        print("\n[Cycle " + str(cycles) + ": EXECUTE Completed]")
        print_registers()
        if not halt:
            input("press Enter for next cycle")
        cycles += 1
    print("       FINAL PROCESSOR STATE")
    print_registers()
run()
```

Execution Results :

The following screenshots illustrate the contents of the various architectural registers after executing all the input programs sequentially.

```
arushjain@Arushs-MacBook-Air IAS_2 % python ias.py
PC  : 000000000110
AC  : 00000000000000000000000000000001101
MQ  : 00000000000000000000000000000000000
IR  : 11111111
MBR : 11111111000000000000000000000000000
MAR : 000000000000
arushjain@Arushs-MacBook-Air IAS_2 %
```

Figure 2: Result

References

1. Wikipedia IAS machine.
2. William Stallings, Computer Organization and Architecture: Designing for Performance.
3. M. Morris Mano, Computer System Architecture.