Basic CPU Components

ALU (arithmetic Logic Unit): Performs arithmetic and logical operations.

Implementation:

.Creat functions for basic operations: add(x,y), subtract(X,Y), and(x,y),etc.

.Updates flags after each operation (E>G., Zero Flag if the result is 0).

Example:

```
Def add(x,y):
  result = x+y
  flags["Z"] = (result == 0)
   return result
```

Implements general-purpose register

To implement general-purpose registers for a basic CPU, follow these steps:

1. Define Registers

General-purpose registers (e.g., R0, R1, R2) are used to store temporary values during program execution.

Use a dictionary or list to represent these registers in your CPU emulator.

2. Basic Structure

```
# Define Registers
registers = {
   "R0": 0,  # General-purpose register 0
   "R1": 0,  # General-purpose register 1
   "R2": 0,  # General-purpose register 2
   "R3": 0,  # General-purpose register 3
}
```

3. Register Operations

Provide basic functions to interact with registers:

a. Read a Register

```
def read_register(register_name):
    return registers.get(register_name, None)
```

b. Write to a Register

```
def write_register(register_name, value):
    if register_name in registers:
        registers[register_name] = value & 0xFF  # Ensure 8-bit values
    else:
        raise ValueError(f"Register {register_name} does not exist.")
```

c. Reset All Registers

```
def reset_registers():
    for reg in registers:
        registers[reg] = 0
```

4. Example Usage

```
# Write to registers
write_register("R0", 42)
```

```python
write_register("R1", 255)

# Read from registers
value = read_register("R0")
print(f"Value in R0: {value}")

# Reset all registers
reset_registers()
print(f"Registers after reset: {registers}")
```

5. Integration with CPU

Registers are central to CPU operations. For example:

Fetching Data from Memory to Register:

```python
def load_to_register(register_name, memory_address, memory):
    value = memory[memory_address]
    write_register(register_name, value)
```

Performing an ALU Operation and Storing the Result:

```python
def execute_addition():
    # Example: ADD R0, R1 (R0 = R0 + R1)
```

```
A = read_register("R0")
B = read_register("R1")
result = A + B
write_register("R0", result)
```

To create the Program Counter (PC) and Instruction Register (IR) for basic CPU components in a concise manner:

1. Components Overview:

Program Counter (PC): Keeps track of the address of the next instruction.

Instruction Register (IR): Holds the current instruction being executed.

Here is a C++ implementation of the Program Counter (PC) and Instruction Register (IR) for basic CPU components:

C++ Code:

```cpp
#include <iostream>
#include <vector>

using namespace std;
```

```
int PC = 0;
int IR = 0;

vector<int> memory = {0x01, 0x02, 0x03, 0xFF}; // Example: ADD, SUB,
JMP, HALT

void incrementPC() {
    PC++;
}

// Fetch instruction from memory
void fetch() {
    if (PC < memory.size()) {
        IR = memory[PC];
        incrementPC();
    } else {
        cerr << "Program Counter out of bounds!" << endl;
        exit(1);
    }
}

bool execute() {
```

```
    switch (IR) {
        case 0x01: // ADD
            cout << "Executing ADD" << endl;
            break;
        case 0x02: // SUB
            cout << "Executing SUB" << endl;
            break;
        case 0x03: // JMP
            cout << "Jumping" << endl;
            break;
        case 0xFF: // HALT
            cout << "Halting program" << endl;
            return false;
        default:
            cout << "Unknown instruction" << endl;
    }
    return true;
}

// CPU cycle: fetch -> decode -> execute
void cpuCycle() {
    bool running = true;
```

```
    while (running) {
        fetch();  // Fetch instruction
        cout << "Fetched instruction: " << IR << " at PC: " << PC - 1 << endl;
        running = execute();  // Execute instruction
    }
}

int main() {
    cpuCycle();  // Run the CPU cycle
    return 0;
}
```

Explanation:

1. Program Counter (PC): Points to the address of the next instruction.
2. Instruction Register (IR): Holds the instruction fetched from memory.
3. Memory: Simulated as a vector of instruction opcodes.
4. CPU Cycle: Fetch the instruction, decode it, and execute it in a loop.

How It Works:

Fetch: Loads the instruction from memory into IR.

Execute: Decodes the instruction and executes corresponding operation.

PC: Increments after each instruction fetch, pointing to the next address.