Name: Fadhla Mohamed

Sirname: Mutua

Matricola: SM3201434

Important

This project was done using windows as such uses the use of windows sys for command-line arguments

main.py

This file is the entry point of the application. It performs the following functions:

- **Argument Parsing**: Uses the `sys` module to parse command-line arguments. It expects at least one argument (the file path) and optionally additional numeric inputs.
- **Usage Validation**: If insufficient arguments are provided, it prints a usage message and raises an exception.
- **File Processing**: Retrieves the file path from the arguments and uses the `unwrap` method from the `Utility` class (imported from `utils`) to read and preprocess the file contents (removing comments, extra spaces, etc.).
- **Input Conversion**: Converts any additional command-line arguments into integers, which are later used as input data for the simulation.
- **LMC Initialization and Execution**: Creates an instance of the `LMC` class (imported from `cpu`), then calls its `run` method, passing in the processed file data and input data.

Overall, `main.py` sets up the environment, processes inputs, and starts the simulation of the Little Man Computer.

cpu.py

This file defines the `LMC` class, which is the core simulation engine for the Little Man Computer. Key aspects include:

- **Inheritance**: `LMC` inherits from the `Assembler` class, gaining the ability to convert assembly code into machine code.
- **Initialization**: Sets up registers and memory. It creates a memory array of 100 slots, initializes the program counter, accumulator, input/output queues, a flag for overflow/underflow, and an instance of the `Utility` class.
- **Instruction Loading**: Implements `load_instructions`, which calls the inherited `assemble_manually` method to translate assembly instructions.
- **Execution Loop**: The `run` method enters a loop where it:
  - Fetches the current instruction from memory, ensuring it is a three-digit string.
  - Decodes the opcode and address.

    - Uses a pre-defined list of instruction functions (sourced from `Utility`) to execute the instruction.
    - Adjusts the program counter and handles special cases like input (`opcode 901`) and output (`opcode 902`).
    - Breaks the loop when a halt instruction is encountered.
- **State Management**: Includes methods for resetting the machine state and printing the exit status.

`cpu.py` is thus responsible for simulating instruction execution in the Little Man Computer model.

utils.py

This file defines the `Utility` class, which contains helper functions for file processing and executing specific LMC instructions:

- **File Unwrapping**: The `unwrap` function reads an assembly file, strips whitespace, removes comments (anything following `//`), converts lines to uppercase, and returns a list of clean instructions.
- **Instruction Functions**: Implements methods corresponding to LMC operations:
    - **Arithmetic**: `lmcAdd` and `lmcSub` perform addition and subtraction.
    - **Memory Operations**: `lmcSTA` and `lmcLDA` store to and load from memory.
    - **Control Flow**: `lmcBRA`, `lmcBRZ`, and `lmcBR` handle unconditional and conditional branching.
    - **I/O Operations**: `lmcINP` fetches input from the input queue (with safeguards against empty queues), and lmcOUT sends data to the output queue and prints it.
    - **Halting and Error Handling**: `lmcHLT` stops execution, and `lmcError` handles invalid opcodes.
    - **Flag Checking**: `check_flag` ensures the accumulator remains within valid bounds.

assembler.py

This file contains the `Assembler` class, which provides functionality for converting assembly code into machine code:

- **Two-Pass Assembly Process**:
    1. **Label Resolution**: The first pass iterates through each line of assembly code to identify and record label names along with their corresponding memory addresses.
    2. **Instruction Translation**: In the second pass, the assembler processes each line to:
        - Remove any labels.
        - Identify instructions (or the `DAT`) and convert them into a machine code format.
        - For standard instructions, it retrieves the opcode from a predefined instruction set, verifies the presence of an operand, resolves numeric addresses or label references, and computes the final instruction as `opcode * 100 + address`.
- **Memory Storage**: The resulting machine code is stored sequentially in the machine's memory array. This module abstracts the complexities of assembly language translation, making it available to the LMC simulation through inheritance.

Connections

The application is orchestrated by `main.py`, which acts as the entry point. It parses command-line arguments to determine which assembly file to process and any input data to be used during simulation.

1. **File Processing**:

    - `main.py` uses the `unwrap` function from `utils.py` to read and preprocess the assembly file, ensuring consistent formatting (e.g. uppercase, no comments).
2. **Assembly**:
    - The `LMC` class in `cpu.py` inherits from `Assembler` (defined in `assembler.py`). When `run` is invoked, `LMC` calls `assemble_manually` to convert the assembly code into machine code, filling the LMC's memory.
3. **Simulation Execution**:
    - `cpu.py` contains the execution loop that fetches instructions from memory and decodes them. It then delegates execution to specific functions defined in the `Utility` class (`utils.py`), which implement the actual LMC operations (arithmetic, memory access, branching, I/O, etc.).
    - Input data is managed via a deque (queue) in `LMC`, ensuring safe and orderly processing of input instructions.
4. **Output and Cleanup**:
    - Throughout the simulation, operations such as output are printed and stored, and upon encountering a halt instruction, the simulation ends. The final state is then printed, and the machine is reset.

Together, these modules read an assembly program, translates it into machine code, and simulates the execution of a Little Man Computer