Design Notes:

(SIMILAR TO PROJECT PART 1 DESIGN NOTES)
Major Updates/Changes:
- All CPU instruction execution logic completed
- User Keyboard input for integer input available on UI
- PC register updated
- Added Cache

1. Assembler Class

The function of Assembler Class is to take an assembly source file as the input and output the
listing file and load file. - Mnemonic to Binary Opcode Mapping
Utilizes a static map, 'mnemonicBinaryOpcodeMapper', for lookup of binary opcodes corresponding to assembly mnemonics. - Single Assembly Instruction Conversion

The 'binMachineCodeConvertedFromAssembly' method handles the conversion of assembly
instructions into binary machine code. - Listing File & Load File Generation
The 'assembly' method reads assembly source code, processes each instruction, and generates a listing file and a load file.

2. BaseConversion Class

The BaseConversion class offers a set of utility methods for converting between different positional notations, including decimal, binary, and octal representations. These methods facilitate numerical conversions for various applications.

Utility Methods:

decimalToBinary(String decStr, int binStrLen):
Converts a decimal string to a binary string with a specified length, adding leading zeros if necessary.

binaryToOctal(String binStr, int octStrLen):
Converts a binary string to an octal string with a specified length, managing leading zeros and grouping binary digits into octal digits.

hexToBinary(String hexNumber):
Converts a hexadecimal number (represented as a string) to its binary equivalent, utilizing hexDigitToBinary(char hexDigit) internally.

octalToBinary(String octalString):
Converts an octal string to a binary string, padding and truncating the result to ensure a maximum length of 16 bits.

decimalToOctal(int x, int octStrLen):
Converts a decimal integer to an octal string with a specified length, ensuring leading zeros are added as needed.

octalToDecimal(String octalStr):
Converts an octal string to a decimal integer representation.

decimalTo6DigitOctal(int decimal):
Converts a decimal integer to a 6-digit octal string, padding with leading zeros if necessary.

binaryToDecimal(String binaryStr):
Converts a binary string to its corresponding decimal integer representation.

3. CPU Class

Method: getBaseInstruction(String instruction)

This method takes an instruction string as input and performs the following steps to determine and return the corresponding base instruction:

Split Instruction: Parses the input instruction string into two parts: address and data.
Convert Address and Data: Converts the address and data parts from octal representation to decimal integers.

Convert Data to Binary:
Converts the data part from octal to binary representation and ensures it is represented as a 16-bit binary string.

Extract Opcode:
Retrieves the first 6 bits of the binary data string, which represent the opcode.

Lookup Base Instruction:
If the opcode corresponds to an instruction, return the determined base instruction name along with the binary data representation, or an error message for invalid instruction formats.

main Method
The main method demonstrates the usage of the getBaseInstruction method with test cases (instruction1, instruction2, instruction3, instruction4). It calls getBaseInstruction for each test case and prints the result.

Notes:
The getBaseInstruction method demonstrates parsing and processing of instruction strings to determine base instructions.
Adjustments or enhancements can be made to handle additional instruction formats or error cases based on specific requirements.

The CPU class provides a foundation for simulating instruction processing within a CPU architecture, focusing on opcode lookup and instruction interpretation based on provided data.

4. CPU_executions Class

Method: perform_action(String dataStr)
This method is responsible for executing instructions based on the provided base instruction and associated parameters.

Input:
dataStr: A string representing the instruction and associated parameters in the format "instruction|parameters", where:
instruction: The base mnemonic code of the instruction (e.g., "LDR", "HLT").
parameters: The binary parameters associated with the instruction.

Execution Logic:
Split Input: Splits the input dataStr into instruction and parameters using the pipe (|) delimiter.

Extract Parameters:
Extracts individual components from parameters:
R: Bits representing the register index (R).
IX: Bits representing the index register specifier (IX).
I: Bit representing the indirect addressing mode (I).
Address: Bits representing the memory address (Address).
Convert Address to Decimal:
Converts the binary representation of Address to a decimal integer (AddressDecimal).
Execute Instructions:
Depending on the instruction, performs specific operations:
HLT: Stops instruction execution.
DATA: Stores data into memory at the specified address.
LDR: Loads a value from memory into a general-purpose register (GPR) based on index registers (IX).
LDA: Loads an address into a general-purpose register (GPR) based on index registers (IX).
LDX: Loads a value into an index register (IX) based on the specified address.

Handle Indirect Addressing (I):
If indirect addressing (I = 1) is specified, retrieves the value from the memory location specified by AddressDecimal.

Set Register Values:
Sets the value retrieved from memory into the appropriate register (GPR or IXR) based on the instruction and specified index register (R or IX).
Output:
Prints relevant debug information during instruction execution.

Available instruction logic:

- LDA (Load Register with Address)

Load a register with a specified address value.
Determine if an index register is used to modify the address.
Set the register destination with the loaded value.

- ORR (Logical Or of Register and Register)

Perform a logical OR operation between two registers.
Retrieve binary values of specified registers.
Compute the OR operation and set the result to specified registers.

- AND (Logical And of Register and Register)

Perform a logical AND operation between two registers.
Retrieve binary values of specified registers.
Compute the AND operation and set the result to specified registers.

- SMR (Subtract Memory from Register)

Subtract a value from a register, taking into account an index register offset.
Retrieve the value from memory and update the register.

- AIR (Add Immediate to Register)

Add an immediate value to a register.
Update the register with the added value.

- TRR (Test the Equality of Register and Register)

Test if two registers contain equal values.
Set condition code bit based on the equality result.

- RFS (Return from Subroutine with Return Code)

Return from a subroutine with a specified return code.
Adjust program execution to the designated return point.

- SIR (Subtract Immediate from Register)

Subtract an immediate value from a register.

Update the register with the result of the subtraction.

- IN (Input Character to Register from Device)

Input a character from a specified device to a register.
Utilize the specified device identifier to perform the input operation.

- JMA (Unconditional Jump to Address)

Perform an unconditional jump to a specified address.
Adjust program execution to the designated target address.

- JNE (Jump if Not Equal)

Conditionally jump to an address based on the equality of two registers.
Evaluate the equality condition and adjust program execution accordingly.

- SETCCE (Set the E Bit of Condition Code)

Set the E bit of the condition code register.
Update the condition code register based on the specified instruction.

- LDR (Load Register from Memory)

Load a register with a value stored in memory at a specified address.

- LDX (Load Index Register from Memory)

Load an index register with a value stored in memory at a specified address.

- STR (Store Register to Memory)

Store the value of a register into memory at a specified address.

- JCC (Jump Conditionally on Condition Code)

Conditionally jump to an address based on the state of the condition code register.

- SOB (Subtract One and Branch)

Decrement a register and branch to a specified address if the result is not zero.

- AMR (Add Memory to Register)

Add a value from memory to a register and update the register.

- STX (Store Index Register to Memory)

Store the value of an index register into memory at a specified address.

- JZ (Jump if Zero)

Conditionally jump to an address if a specified register is zero.

- JSR (Jump to Subroutine)

Jump to a subroutine, saving the return address in a specified register.

<u>Additional Notes</u>

The perform_action method encapsulates the logic for executing different base instructions based on the provided mnemonic and associated parameters.

- Additional instruction cases can be added to handle more instruction types.
- Debug statements are included to provide visibility into the execution flow and values used during instruction processing.
- These design notes outline the functionality and execution flow of the CPUExecutions class, demonstrating how instructions are interpreted and executed within a simulated CPU environment. Adjustments can be made based on specific requirements and to enhance error handling or additional instruction types.

5. Machine Simulator Class

Overview

The MachineSimulator class implements a graphical user interface (GUI) for simulating machine operations. It facilitates loading, storing, and executing instructions based on provided input files. Key components include register displays, binary checkboxes, and control buttons. A User input box that receives user keyboard inputs is also available to intake integers for determining closest integer to data set

Class Structure

Extends: JFrame - Inherit frame properties for the GUI.

Implements: ActionListener - Handles action events from buttons.

Constants
REGISTER_COUNT: Total number of registers displayed.
CHECKBOX_COUNT: Total number of binary checkboxes.

Components
Button Components:
loadBtn, loadPlusBtn, storeBtn, storePlusBtn, runBtn, stepBtn, haltBtn, IPLBtn: Action buttons for loading, storing, and executing instructions.

Label Components:
registerLabels: Labels for register names.
registerValueLabels: Labels displaying register values in binary.
currentHexValue: Label showing the hexadecimal value based on checkbox states.
status: Label to display simulation status.

Checkbox Components:
checkboxes: Binary checkboxes representing values.

Panel Components:
contentPanel: Main panel containing register displays and checkboxes.
buttonPanel: Panel holding action buttons.
checkboxPanel: Panel for displaying binary checkboxes and hex value.

Methods
initComponents(): Initializes GUI components such as labels, buttons, and panels.
addComponents(): Adds initialized components to the frame layout.
addListeners(): Attaches event listeners to buttons and checkboxes.
updateHexValueLabel(): Calculates and updates the hexadecimal value label based on checkbox states.
actionPerformed(ActionEvent e): Handles button click events by implementing action logic.

Additional Features
File Handling:
duplicateFile(String originalFilePath, String duplicateFileName): Duplicates file content.
File loading using JFileChooser for instruction input.

Execution Flow

- main(String[] args): Entry point for the simulation.
- Initializes MachineSimulator and related classes.
- Handles file input and simulates machine operations based on loaded instructions.

UI and Interaction

- Displays register values and binary representations.
- Interacts with checkboxes and buttons to simulate instruction execution.
- Provides status updates during the simulation process.

6. Memory Class

Overview
The Memory class represents a simulated memory unit with functionality to load, store, and display memory contents. It utilizes a LinkedHashMap to manage memory locations and their corresponding values.

Class Structure
Imports: java.util.LinkedHashMap, java.util.Map - Utilizes map-based data structures.
Access Modifiers: public - Allows external access to class methods.

Attributes
memory: Static LinkedHashMap for storing memory contents (<String, String> key-value pairs).
SIZE: Constant representing the size of the memory (default: 4096).

Constructor
Memory(): Initializes the memory by populating it with default values ("000000" for each location).

Methods

- load(String mar):
- Converts the memory address (mar) to octal format.
- Retrieves the value from memory using the octal address.
- Returns the memory value or null if the address does not exist.
- store(String mar, String mbr):
- Stores a value (mbr) at the specified memory address (mar).
- showMemoryContents():
- Displays the contents of the memory (LinkedHashMap entries).

7. Registers Class

Overview
The Registers class represents a set of simulated processor registers used in machine simulation. It provides methods to set and retrieve values for different types of registers.

Class Structure
Access Modifier: public - Allows external access to class methods.

Attributes: Private static fields representing different types of registers.

Attributes

General Purpose Registers (gpr0 to gpr3):
Stored as strings representing binary values (16-bit).
Special Registers:
ir (Instruction Register)
mar (Memory Address Register)
mbr (Memory Buffer Register)
mfr (Machine Fault Register)
msr (Machine Status Register)
pc (Program Counter)
ixr1, ixr2, ixr3 (Index Registers)
cc (Condition Code)

Constructor
Registers(): Initializes all registers to default values ("0000000000000000").

Methods
setRegisterValue(String registerName, String value):
Sets the value of the specified register based on the provided register name.
Uses a switch statement to determine which register to update.
getRegisterValue(String registerName):
Retrieves the value of the specified register based on the provided register name.
Uses a switch statement to determine which register value to return.

Error Handling
If an invalid register name is provided (default case in switch), an error message is displayed, and a default value ("-1") is returned.

Usage
Setting Register Values:
Utilizes setRegisterValue method to update the values of different registers.
Provides flexibility to modify specific registers based on their names.

Getting Register Values:
Utilizes getRegisterValue method to retrieve the values of different registers based on their names.
Enables access to register values for various operations within the simulation.

8. Parse Octal File Class:

Overview

The ParseOctalLoadFile class is designed to parse a text file containing octal load pairs (key-value pairs) and store them in a LinkedHashMap for further processing.

Class Structure

Access Modifier: public - Allows external access to class methods.
Attributes:

octalLoadPairs: A LinkedHashMap to store octal load pairs parsed from the file.

Constructor

- ParseOctalLoadFile(File loadFile):
- Initializes octalLoadPairs.
- Reads and processes the contents of the specified file (loadFile).
- Parses each line of the file, expecting a key-value format separated by whitespace.
- Populates the LinkedHashMap (octalLoadPairs) with the parsed key-value pairs.

Methods

getOctalLoadPairs() : LinkedHashMap<String, String>:
Returns the LinkedHashMap containing the parsed octal load pairs.
Allows external access to the parsed data for further use.

File Parsing

Reading File:

- Uses BufferedReader to efficiently read the file line by line.
- Handles file I/O operations within a try-with-resources block to ensure proper resource management.

Parsing Lines:

- Each line is trimmed and split using regular expressions (\\s+) to handle multiple spaces as separators.
- Expects lines to contain exactly two parts (key and value).
- Inserts valid key-value pairs into the LinkedHashMap.

Error Handling

IOException:
Catches and prints stack trace for any I/O errors that occur during file reading.
Ensures graceful error handling and notification of file processing issues.

Usage

Instantiation:
Creates an instance of ParseOctalLoadFile by providing a File object representing the file to be parsed.

Accessing Parsed Data:

Utilizes getOctalLoadPairs() method to retrieve the parsed octal load pairs stored in the LinkedHashMap

9. Cache Class

The Cache class provides functionality for a 16-block fully associative unified cache within the Memory Control Unit (MCU) of a simulated system. This design aims to efficiently manage memory access through caching mechanisms.

Class Structure:

Fields:

cacheLines: List of cache lines, representing the cache's content.

Constructor:

Initializes the cache with empty cache lines.

Methods:

- containsAddress(int address):
- Checks if the cache contains a specific memory address by comparing against stored cache lines' tags.
- Returns a boolean indicating presence in the cache.
- add(int address, int value):
- Adds a new memory address and associated value to the cache.
- Implements replacement policy (e.g., FIFO) if the cache is full.
- fetchFromCache(int address):
- Retrieves a word from the cache based on the given memory address.
- If the address is found in the cache, returns the corresponding value directly.
- If not found, fetches the value from main memory, updates the cache, and returns the fetched value.
- storeIntoCache(int address, int value):
- Stores a value into both the cache and main memory at the specified address.
- Implements replacement logic to manage cache size constraints.

Cache Line (CacheLine):

Fields:

- tag: Memory address tag associated with the cache line.
- data: Data stored in the cache line.

Memory Control Unit (MCU) Integration:

Initialization:
The MCU class instantiates a Cache object as part of its memory management components.

Cache Usage:
MCU methods utilize the cache for efficient memory access:
fetchFromCache: Fetches data from the cache if available, falling back to main memory if necessary.

storeIntoCache: Stores data into the cache and main memory simultaneously, managing cache replacement.

Summary Notes

The classes Memory, Registers, and ParseOctalLoadFile interact with each other to form the foundation of a virtual machine.

Memory Class:
The Memory class represents the virtual memory of the machine, initialized as a LinkedHashMap in which memory locations and their corresponding values are stored. It provides methods like load and store to interact with memory. The load method retrieves a value from memory based on a given memory address (MAR), while the store method updates a memory location with a specified value.

Registers Class:
The Registers class encapsulates various registers of the virtual machine, such as general-purpose registers (GPRs), instruction register (IR), memory address register (MAR), memory buffer register (MBR), program counter (PC), and others.
It offers methods to set and get values for these registers dynamically. For instance, setRegisterValue modifies the value of a specified register, and getRegisterValue retrieves the value of a given register.

ParseOctalLoadFile Class:

- The ParseOctalLoadFile class is responsible for reading an external file containing octal load pairs and parsing them into a LinkedHashMap structure.
- It initializes an instance of LinkedHashMap (octalLoadPairs) to hold the parsed key-value pairs.

-
- This class plays a role in loading initial values into the virtual machine's memory by populating the Memory class with data parsed from the external file.

Initialization:

- Upon instantiation of the virtual machine, an instance of Memory is created, which initializes the memory with default values.
- Similarly, Registers are initialized with default values through its constructor.

Loading Initial Data:

- The ParseOctalLoadFile class is invoked to parse an external file containing initial memory values.
- The parsed data (octal load pairs) is stored in the Memory class using its store method, populating the virtual memory with the initial values.

Execution:

- During the execution of the virtual machine, the Registers class manages the state of various machine registers, updating them as needed based on instructions being executed.
- The Memory class is accessed by other components (e.g., instruction execution logic) to load instructions or store data during program execution.

Overall Flow:

- The virtual machine's operation involves fetching instructions and data from memory using the Memory class and executing them while maintaining state in the Registers class.
- The ParseOctalLoadFile class facilitates the initialization of memory with initial values from an external source, contributing to the startup and configuration of the virtual machine before program execution begins.