Class Definitions

Class Summaries

Class Program: `class Program`

Purpose: The program class is the driver for the application. It contains the main method entry point for the application.

Method BuildAvaloniaApp: `public static AppBuilder BuildAvaloniaApp`

Purpose: Configures settings for the Avalonia based GUI, and returns an AppBuilder object, which is used to display the main window and run the application.

Parameters: None

Return: An AppBuilder object.

Method Main: `public static void Main`

Purpose: Calls BuildAvaloniaApp() and uses the returned AppBuilder object to launch the GUI application.

Parameters: string[] args, default list of command line arguments, none required for this application.

Return: Void

Class OperatingSystemGUI: `public class OperatingSystemGUI`

Purpose: The OperatingSystemGUI class connects all the components of the virtual machine and exposes them to the view model. Exposure to the view model allows the components to update the GUI and be manipulated by the user through the GUI.

Constructor OperatingSystemGUI: `public OperatingSystemGUI`

Purpose: Creates an OperatingSystemGUI object with a Processor object named CPU and a Memory object named MainMemory. Links MainMemory to CPU.

Method Execute: `public void Execute`

Purpose: Initiates execution of the instructions stored in MainMemory beginning at location 0 by calling the Execute method of the Processor class.

Parameters: None

Return: Void

Enum BasicML: `public enum BasicML`

Purpose: The BasicML enumerator maps the instructions of the BasicML language to their numerical values.

Enum BranchCondition: `public enum BranchCondtion`

Purpose: The BranchCondition enumerator maps the three different branch conditions to the numerical values of their corresponding instructions.

Class Memory: `public class Memory`

Purpose: The Memory class models the memory of the virtual machine. It contains a nested class, MemoryLine (see **Nested Class MemoryLine**), which models each location in memory in a way which the GUI can interpret. A memory object contains an ObservableCollection of MemoryLine objects named locations which represents all locations in memory. The MaxWord field of the Memory object dictates the largest absoulte data value a location can hold. The Capacity field dictates how many MemoryLine objects the locations collection can hold.

Constructor Memory: `public Memory`

Purpose: Creates a Memory object and initiates all locations to hold a MemoryLine object with data value of zero.

Parameters: int capacity: An integer that represents the Capacity of the memory, the default value is 100. Int maxWord: An integer that represents the maximum absolute data value that can be store in a MemoryLine, the default value is 9999.

Method Read: `public int Read`

Purpose: Returns the data value stored at a specified memory location.

Parameters: int location: An integer representing a memory location

Return: An integer which is the data word at the memory location. If location is invalid, it returns 0.

Method WriteWord: `public bool WriteWord`

Purpose: Writes a word of data to the specified memory location.

Parameters: int location: An integer representing a memory location. int data: an integer representing the data value to write.

Return: A Boolean representing success or failure. Writes and returns true if location is valid and data is valid, Does not write and returns false otherwise

Method WriteFile: `public bool WriteFile`

Purpose: Writes a file to memory line by line, starting at the specified location.

Parameters: int location: An integer representing a memory location. string filename: Path to a file with contents to write to memory.

Return: A Boolean representing success of failure. True if successful, false otherwise.

Pre-condition: The file must contain text data with each line containing a sign (+ or -) and a 4 digit decimal word.

Post-condition: Any line which does not match the described format will be skipped, and all others will be written to the memory.

Method SaveToDisk: `public void SaveToDisk`

Purpose: Save the contents of all memory locations to a file.

Parameters: string filename: Path to the file to create if nonexistent or overwrite if it already exists.

Return: None

Post-condition: The file will contain the data value of all MemoryLines, even those that are zero, each written on their own line in the file. The file is therefore a snapshot of the state of the virtual machine's memory. The data on each line contains a sign (+ or -) and a 4 digit decimal word.

Nested Class MemoryLine: `public class MemoryLine : ObservableObject`

Purpose: The MemoryLine class formats the data stored at each memory location as well as information about that data. It has a property LineNumber that stores the index of the location, Data property which stores the signed 4 digit decimal word at the location, Instruction property which interprets the first two digits of the data word and translates it to the possible corresponding instruction. It contains no methods. The MemoryLine class inherits from the ObservableObject class which allows the properties to be used to auto update the GUI if they are changed.

Class Register: `public class Register`

Purpose: The Register class models a register which can store one signed four digit word of data for the processor. The Register object contains a public Data property which can be directly accessed by the Processor object. It contains no methods.

Constructor Register: `public Register`

Purpose: Creates a Register object and initiates its Data property to 0.

Class Processor: `public class Processor`

Purpose: The Processor class models the processor of the virtual machine. The processor class has a Register object named accumulator. The processor class also has a Memory object named mainMemory. Additionally it contains a field, currentLocation which contains the index of the next instruction to be executed.

Constructor Register: `public Processor`

Purpose: Creates a Processor object and initializes its mainMemory field to the specified Memory object.

Parameters: Memory mainMemory: The memory object that the processor will use for reading and writing.

Method: `public int GetCurrentLocation`

Purpose: Returns the current memory location where the processor will look for the next instruction.

Parameters: None

Return: An integer representing the current memory location.

Method: `public void SetAccumulator`

Purpose: Sets the data word stored in the accumulator to the specified value.

Parameters: int value: The value to store in the accumulator.

Return: None

Method: `public int GetAccumulator`

Purpose: Returns the current data word stored in the accumulator.

Parameters: None

Return: An integer representing the value of the data word stored in the accumulator.

Method: `public bool Execute`

Purpose: Resets the accumulator and begins executing instructions from the specified memory location. Execution continues until either the end of memory is reached, or a Halt instruction is encountered.

Parameters: int location: The memory location to begin execution from.

Return: A boolean representing success or failure. Returns true after execution if the starting location is valid, false otherwise.

Pre-condition: The specified location must be within the bounds of memory.

Post-condition: The currentLocation field will always be equal to the memory capacity after execution, whether execution ends due to end of memory, or a HALT instruction being reached.

Method: `public bool Interpret`

Purpose: Updates current location to point to the next memory location to look for instructions. Then executes the instruction that was stored at the original location by calling the appropriate method based on the instruction digits.

Parameters: None

Return: A boolean representing whether the instruction was executed successfully. Returns true for valid instructions and false if an invalid instruction code is encountered.

Method: `public void Read`

Purpose: Reads a word of data from the keyboard and stores it in the specified memory location.

Parameters: int location: The memory location where the input word will be stored.

Return: None

Pre-condition: The user must input a valid 4-digit decimal number.

Method: `public void Write`

Purpose: Writes the word stored at the specified memory location to the output window.

Parameters: int location: The memory location of the value to be output.

Return: None

Method: `public void Load`

Purpose: Loads a word from the specified memory location into the accumulator.

Parameters: int location: The memory location to load from.

Return: None

Method: `public void Store`

Purpose: Stores the data currently in the accumulator into the specified memory location.

Parameters: int location: The memory location where the value from the accumulator will be stored.

Return: None

Method: `public void Add`

Purpose: Adds the word at the specified memory location to the value in the accumulator and leaves the sum in the accumulator.

Parameters: int location: The memory location whose value will be added to the accumulator.

Return: None

Method: `public void Subtract`

Purpose: Subtracts the word at the specified memory location from the value in the accumulator and leaves the difference in the accumulator.

Parameters: int location: The memory location whose value will be subtracted from the accumulator.

Return: None

Method: `public void Divide`

Purpose: Divides the value in the accumulator by the word at the specified memory location leaves the quotient in the accumulator.

Parameters: int location: The memory location whose value will be used to divide the accumulator's value.

Return: None

Method: `public void Multiply`

Purpose: Multiplies the value in the accumulator by the word at the specified memory location leaves the product in the accumulator.

Parameters: int location: The memory location whose value will to multiply with the accumulator's value.

Return: None

Method: `public void Branch`

Purpose: Changes the currentLocation to the specified memory location if the specified condition is met.

Parameters: int condition: The instruction code that determines whether branching occurs. int location: The memory location to branch to if the condition is met.

Return: None

Method: `public void Halt`

Purpose: Halts the execution of the program by setting the current location to the memory's capacity. This triggers the execute loop to cease.

Parameters: None

Return: None

Post-condition: Execution is halt and no further instructions are executed. The currentLocation field is equal to the memory's capacity.

Class App: `public partial class App : Application`

Purpose: The App class inherits from the Application class which encapsulates the Avalonia application. It manages application functionality, such as styles and templates.

Method: `public override void Initialize`

Purpose: Loads the .axaml files into the GUI application.

Parameters: None

Return: None

Method:

`public override void OnFrameworkInitializationCompleted`

Purpose: Manages the lifetime of the application’s main window

Parameters: None

Return: None

Class MainWindow: `public parital class MainWindow : Window`

Purpose: The MainWindow class represents the main window of the GUI application. It inherits from the Avalonia.Controls.Window class. It contains a ViewModel object named ViewModelData, which contains the resources the GUI will access.

Constructor MainWindow: `public MainWindow`

Purpose: Creates the MainWindow object as a GUI component. Initializes the ViewModelData and sets it as the data context for finding resources.

Method: `public aysnc void OnUpload_Click`

Purpose: When the user clicks the upload button, this method launches a dialog window which prompts the user to select a filepath for upload.

Parameters: object sender: the object that raised the event, RoutedEventArgs args: data specific to the raised event

Return: None

Method: `public aysnc void OnSave_Click`

Purpose: When the user clicks the save button, this method launches a dialog window which prompts the user to select a filepath to save the file.

Parameters: object sender: the object that raised the event, RoutedEventArgs args: data specific to the raised event

Return: None

Method: `public void Window_Resized`

Purpose: Ensures the window components resize appropriately with a change in the window size.

Parameters: object sender: the object that raised the event, RoutedEventArgs args: data specific to the raised event

Return: None

Method: `public aysnc void OnRun_Click`

Purpose: When the user clicks the run button, the CPU executes the instructions in memory starting at location 0. Once execution is complete, the view's register property is update with the final register value.

Parameters: object sender: the object that raised the event, RoutedEventArgs args: data specific to the raised event

Return: None

Method: `private void Input_KeyDown`

Purpose: Only accepts numerical entries in the input field.

Parameters: object? sender: the object that raised the event. Avalonia.Input.KeyEventArgs e: represents the key that was pressed.

Return: None

Method: `private void NumericTextBox_PropertyChanged`

Purpose: Ensures that only appropriate data values are entered in the textboxes.

Parameters: object? sender: the object that raised the event. AvaloniaPropertyChangedEventArgs e: information about the property that was changed.

Return: None

Nested Class ViewModel:

`public partial class ViewModel : ObservableObject`

Purpose: The ViewModel class represents the model data that is exposed to the view (the GUI application). This is where the virtual machine is initialized. It contains properties which are observable. Observable properties automatically update the view when their data changes.

Constructor ViewModel: `public ViewModel`

Purpose: Creates the ViewModel object and initializes its resources. The constructor creates the virtual machine and exposes the virtual machine's memory and register to the view (the GUI).