CPU: The purpose of the CPU class is to execute instructions based on the memory loaded from the file path the user inputs.

Attributes

Memory -> type Memory: This is a parameter when creating a CPU Object. It is an object taken from the memory class.

Accumulator type -> accumulator: this is a parameter when creating a CPU Object. It is an object taken from the accumulator class.

Instruction Counter type -> int: This is a counter so that the cpu knows when the memory runs out of space

Methods

+ handle\_read(address): This is a method for handling the read function. It prompts the user for a word to be stored into memory. Then stores said the user's word at the given address in the word used when calling the function. NO return value, no prerequisites. Post requisite: The users word they inputted is stored in the memory address that was a parameter in the function call.

+ handle\_write(address): This is a method for handing the write function. It will pull from the memory at the given address, and write that word to the screen. No return value. Prerequisite: at the address given when calling the function, there is already a word there. Post requisite: The word at the given address is written to the terminal

+ handle\_load(address) This is a method for handling the load function. It will take a word from a specific location in memory, and put that word into the accumulator. No return value. Prerequisite: at the address given when calling the function, there is already a word there. Post-requisite: the word at the given address is loaded into the accumulator’s value.

+ handle\_add(address) This is a method for handling the add function. It will take a word from a specific location in memory, and add the word to the word in the accumulator(if a value is there).Return type: None. Prerequisite: at the given address, there is a word there. Post-requisite: The pre accumulator value + value in the address given when calling the function = post accumulator value

+ handle\_subtract(address): This is a method for handling the subtract function. It will take a word from a specific location in memory, and subtract the word from the word in the accumulator(if a value is there).Return type: None. Prerequisite: at the given address, there is a word there. Post-requisite: The pre accumulator value - value in the address given when calling the function = post accumulator value

+ handle\_multiply(address): This is a method for handling the multiply function. It will take a word from a specific location in memory, and multiply the word to the word in the accumulator(if a value is there).Return type: None. Prerequisite: at the given address, there is a word there. Post-requisite: The pre accumulator value \* value in the address given when calling the function = post accumulator value

+ handle\_divide(address): This is a method for handling the divide function. It will take a word from a specific location in memory, and divide the word from the word in the accumulator(if a value is there).Return type: None. Prerequisite: at the given address, there is a word there. Post-requisite: The pre accumulator value / value in the address given when calling the function = post accumulator value

+ handle\_store(address): This is a method for handling the store function. It takes a word from a specific location in memory, and stores it into the accumulator. Return type: None. Pre-requisites: None. Post requisites: The word at the given address is now stored into the accumulator.

+ handle\_branch(address): This is a method for handing the branch function. It branches to a word in memory using the cpu’s instruction counter attribute. Return type: None. Prerequisite: There is a word at the specified address. Post-requisite: The instruction counter == address given at the function call.

+ handle\_branchNeg(address): This is a method for handling the branchNeg function. It branches to a word in memory using the cpu’s instruction counter attribute if the accumulator value is negative. Return type: None Prerequisite: There is a word at the specified address. Post-requisite: The instruction counter == address given at the function call if the accumulator value is negative.

+ handle\_branchZero(address): This is a method for handling the branchZero function. It branches to a word in memory using the CPU's instruction counter if the accumulator value == 0. Return type: none. Prerequisite: There is a word at the specified address. Post-requisite: The instruction counter == address given at the function call, if the accumulator value == 0.

+ handle\_halt(): This is a method for handling the halt function. It sets the instruction counter at zero, stopping the program from reading more words in memory. Return type: 0. Prerequisite: None. Postrequisite: function stops, instruction counter == 100

+execute\_instruction(): This is a method for calling the other functions and reading the instruction from memory. It first gets the instruction, then separates it into two values Operator, and operand. Depending on the operator, it will call either of the functions listed above, and will pass the operand as a parameter into the functions(besides halt). Return type: 0. Prerequisite: None. Postrequisite: The word from memory is properly executed, or ignored if the word is invalid.

Memory: The purpose of the memory class is to hold the instructions from the user inputted file for machine instructions.

Attributes

memory -> type list: This is a value initially created when creating a class. It is blank at first.

Methods

+get\_value(address): This is a method for getting a specific value from the address given when calling this function. Return type: int Prerequisite: None Post requisite: at the address given when calling the function, that value is returned.

+set\_value(address, value): This is a method for setting a specific value from the address given when calling this function. Return type: int Prerequisite: None Post requisite: at the address given when calling the function, the inputted value is set into that location in memory.

Accumulator: The purpose of the accumulator class is to have a a part where you can modify it, to calculate inputs.

Attributes

Value -> type int: This is the value of the accumulator. It is 0 at first.

Methods

+add\_value(value): This is a method for adding a value passed by parameter into the function. Return type -> None. prerequisite: None. Post requisite: the accumulator pre value + value passed into function == accumulator post value

+Subtract\_value(value): This is a method for subtracting a value passed by parameter into the function. Return type -> None. prerequisite: None. Post requisite: the accumulator pre value - value passed into function == accumulator post value

UVUSimApp: this class under GUI.py handles how the UXI will run, how the user can customize the color scheme for the UI, and save/load their program files.

### **Attributes**

* **cpu** (CPU):  
  The core processor where all calculations are handled.
* **memory** (Memory):  
  Storage for all instructions and data.
* **input\_handler** (GUIInputHandler):  
  Manages user input through the GUI.
* **is\_loaded** (bool):  
  Checks if the program file has been loaded into the current instance.
* **main\_color** (list):  
  The primary color used for the UI theme.
* **off\_color** (list):  
  The secondary color used for the UI theme.
* **main\_layout** (BoxLayout):  
  The main layout container for arranging all UI elements.
* **machine\_instructions\_input** (TextInput):  
  A text field for users to enter or load machine instructions.
* **console\_input** (TextInput):  
  A single-line input field for user commands.
* **output\_display** (TextInput):  
  A read-only field that displays program output and messages.
* **load\_button** (Button):  
  Button to load or reload a program into memory.
* **run\_button** (Button):  
  Button to start execution of the loaded program.
* **save\_button** (Button):  
  Button to save the current program state or instructions to a file.
* **pick\_file\_button** (Button):  
  Button to open a file chooser and load a program from a file.
* **primary\_color\_input** (TextInput):  
  Field for entering a custom primary color for the UI.
* **off\_color\_input** (TextInput):  
  Field for entering a custom off-color for the UI.
* **submit\_color\_input\_button** (Button):  
  Button to apply the selected color inputs to the UI theme.

### **Methods**

* **build()**Builds the GUI structure.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: GUI elements are displayed in the application window.
* **load\_program(instance)**Loads the machine instructions from the input field into memory.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Program is loaded into memory, or an error message is displayed if loading fails.
* **run\_program(instance)**Initiates the CPU to start running the loaded program.
  + **Return Value**: None
  + **Preconditions**: A program is loaded in memory.
  + **Postconditions**: CPU executes the program, and results are displayed in the output area.
* **execute\_cpu()**An asynchronous method to execute CPU instructions until program completion.
  + **Return Value**: None
  + **Preconditions**: A program is loaded and ready to execute.
  + **Postconditions**: Instructions are executed, or an error is displayed if an issue occurs.
* **save\_file(instance)**Opens a popup to save the program to a specified file location.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Program file is saved at the specified location or an error message is displayed.
* **confirm\_save(instance)**Confirms the save action and writes program instructions to a file.
  + **Return Value**: None
  + **Preconditions**: save\_file is triggered and a path is provided.
  + **Postconditions**: File is saved at the provided path, or an error message is shown.
* **pick\_file(instance)**Opens a file chooser popup for selecting a program file to load.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Program is loaded from the selected file, or an error message is displayed.
* **pick\_color(instance)**Updates the UI theme colors based on user input.
  + **Return Value**: None
  + **Preconditions**: Primary and/or off-color values are provided by the user.
  + **Postconditions**: UI theme updates with the selected colors, or an error message is displayed if the input is invalid.
* **parse\_color\_input(color\_input)**Parses color input in hex or rgba format for color customization.
  + **Return Value**: List representing [r, g, b, a] or None if invalid.
  + **Preconditions**: Valid hex or rgba color input.
  + **Postconditions**: Parsed color is returned, or an error message is displayed.
* **update\_theme()**Updates button colors and the background based on main\_color and off\_color.
  + **Return Value**: None
  + **Preconditions**: main\_color and off\_color are set.
  + **Postconditions**: Buttons and background color are updated.
* **\_update\_rect(instance, value)**Updates the background rectangle size and position when layout changes.
  + **Return Value**: None
  + **Preconditions**: Called automatically when layout changes.
  + **Postconditions**: Background rectangle matches the layout dimensions.
* **submit\_console\_input(instance)**Handles console input submission, passes it to the CPU.
  + **Return Value**: None
  + **Preconditions**: Valid input text in console\_input.
  + **Postconditions**: Console input is processed by the CPU.
* **enable\_console\_input()**Activates the console input field for user interaction.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Console input is enabled and ready for user input.
* **output\_callback(message)**Receives output from the CPU and displays it in the GUI.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Message is displayed in output\_display.
* **on\_start()**Integrates the asyncio event loop with the Kivy application.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Asyncio loop starts processing events with the application.
* **process\_asyncio\_events(dt)**Processes pending asyncio events in the application.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Asyncio tasks are processed.
* **on\_stop()**Stops the asyncio event loop on application exit.
  + **Return Value**: None
  + **Preconditions**: None
  + **Postconditions**: Asyncio loop stops, and resources are cleaned up.