**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 -> type CPU: This is the brains of the program. Where all the calculations are calculated.
* Memory -> type Memory: This is where all the words for the functions are stored.
* inputhandler -> type GuiInputHandler: This is how the gui is able to handle inputs from the user.
* is\_loaded: checks if the saved files have been loaded into the current program instance.
* main\_color: handles the primary color used for the UI scheme.
* off\_color: handles the secondary color used for the UI scheme.
* Main\_layout
* Machine\_ionstructions\_input
* Console\_input
* Output\_display
* Load\_button:
* Run\_button:
* Save\_button:
* Pick\_file\_button:

Methods:

+build(): This is how the Gui gets built. Return value -> None. Prerequisite: None: Post Requisite: The gui is successfully displayed

+load\_program(instance: This is how the GUI loads the words into memory. Return value-> None Prerequisite: None. Post requisite: The file program is successfully loaded into the program, or a correct error message is displayed if not.

+run\_program(instance): This is the method to call the cpu to start running the program. Return value-> None Prerequisite: None. Post requisite: The Cpu has run the program(with user input if necessary), and values are displayed.

+submit\_console\_input(): This method is used to get the console input from the gui and give it to the cpu. Return value-> None Prerequisite: There was valid text given to the console. PostRequisite: It was successfully given to the CPU.

+enable\_console\_input(): This method is used to to enable the console input field. Return Value -> None Prerequisite: None: Post Requisite: The console is now ready to receive input.

+output\_callback(message): This method receives a message from the cpu and displays it onto the gui. Return Value -> None Prerequisite: None. Post Requisite: the message is now successfully displaying on the GUI

+onstart(): This method is used to integrate asyncio event loop with the GUI. Return value -> none Prerequisite: None. Post requisite: The gui is now synced with the rest of the program.

+process\_asyncio\_events(dt): This method is used to process the events and tasks sent by asyncio. Return value -> None Prerequisite: None. Post requisite: The said tasks have been successfully handled

on\_stop(): This method stops the loop and program. Return value -> none. Prerequisite: None. Post requisite: The loop and program have stopped.