Create a Python code if you choose programming type. You will give an algorithm if you choose written type. Both the Binary Conversion and Storages will be given.

I. Binary Conversion

Filename: bin_convert.py
This file contains two binary conversion classes and a length-fixing class

Class: `Length`: Contains constant lengths and length-related static Methods.

1) Attributes: Contains constant lengths and Methods that add or remove text or decimal places.
   :a) `whole`: The length of the e for the `HalfPrecision` class.
   :b) `fraction`: The length of the f for the `HalfPrecision` class.
   :c) `precision`: The length of the `HalfPrecision` format.
   :d) `instrxn`: The length of the our `ISA Instruction` format.
   :e) `dec_place`: The length of decimal places the `HalfPrecision` will be rounded off to.
2) Constructor: None
3) Static Method: `trimDec`
   :a) Definition: Rounds off a string into `dec_place` decimal places.
   :b) Parameters: `value` and `places` (default value is `dec_place`)
   :c) Process: Converts `value` into a float and round it off to `dec_place` decimal places.
   :d) Returns: The rounded off float.
4) Static Method: `addZeros`
   :a) Definition: Fill zeros to the left or the right of the `value` based on the length specified by `strlen`.
   :b) Parameters: `value`, `strlen` and `lead` (default value is True)
   :c) Process: Convert the value to a binary string if it is not a string. Fill zeros to the left if `lead` is True, fill zeros to the right, otherwise.
   :d) Returns: A string will length of `strlen`.

Class: `BinaryFraction`: Converts fractional decimal to fractional binary and vice versa.

1) Attributes: None
2) Constructor: None
3) Static Method: `idec2bin`
   :a) Definition: Converts a fractional decimal to a fractional binary.
   :b) Parameters: `idec` and `ibinlen` (default value is `fraction` constant of the `Length` class)
   :c) Process: Uses a formula to convert a fractional decimal to a fractional binary and fixes the length equal to `ibinlen`.
   :d) Returns: The fractional binary with length `ibinlen`.
4) Static Method: `ibin2dec`
   :a) Definition: Converts a fractional binary to a fractional decimal.
   :b) Parameters: `ibin`
   :c) Process: Uses a formula to convert a fractional binary to a fractional decimal.
   :d) Returns: The fractional decimal.

Class: `HalfPrecision`: Converts decimal to Half Precision binary format.

1) Attributes: None
2) Constructor: None
3) Static Method: `hpbin2dec`
   :a) Definition: Converts a Half Precision binary format to a decimal.
   :b) Parameters: `binum` and `binlen` (default value is the `whole` constant of the
   :c) Process: Uses a formula to convert a Half Precision binary format to a decimal.
   :d) Returns: The rounded of decimal equivalent.
4) Static Method: `hpdec2bin`
   :a) Definition: Converts a decimal to a Half Precision binary format.
   :b) Parameters: `decnum` and `binlen` (default value is `whole` constant of the `Length` class) `Length` class)
   :c) Process: Uses a formula to find the value of s, e, and f. Concatenate the three values.
   :d) Returns: The Half Precision binary format with length `binlen`.
5) Static Method: `hpbin2bin`
   :a) Definition: Converts a Half Precision binary format to a binary.

:b) Parameters: `bin_str` and `binlen`

:c) Process: Uses the `HalfPrecision` conversation to decimal. Convert the decimal to binary while fixing the length to `binlen`.

:d) Returns: The binary format with length `binlen`.

6) Static Method: `bin2hpbin`

:a) Definition: Converts a binary to a Half Precision binary format.

:b) Parameters: `bin_str`

:c) Process: Converts to decimal. Uses the `HalfPrecision` conversation to binary.

:d) Returns: The binary format.

7) *Also Included*: A tester on how many decimal values are correctly converted by Half Precision format. Don't use it if you don't know how to use it.

II. Storages

Filename: storage.py This file contains a `Storage` class and two storage (memory and register) that mimic the computer storage. Another storage was made to access specialized blocks both in the memory and register easily

Class: `Storage`: Use to store and load data from a storage. It can also display the values of the storage.

1) Attributes: Contains constant lengths and Methods that add or remove text or decimal places.

:a) `data`: A dictionary that contains the value and the key that represents the address.

2) Constructor: Initialize the `data`

:a) Parameters: `data`

:b) Process: Sets the data. It uses `deepcopy` (python built-in) because, for unknown reasons; the memory, register, and special storage have the same values without the `deepcopy`.

3) Class Method: `load`

:a) Definition: Loads the value on the specified `address`.

:b) Parameters: `address`

:c) Process: Gets the value of the `data` using the `address`. The loaded value can be an Instruction format or a Half Precision format.

:d) Returns: The value if it is an Instruction format. Decimal value using Half Precision conversion.

4) Class Method: `store`

:a) Definition: Store or create the `value` on the `data` given the `address`.

:b) Parameters: `value` and `address`

:c) Process: The loaded value can be an Instruction format or a Half Precision format. If the value is a decimal, it converts the value to Half Precision binary format. Finally, store it to the `data` using the `address`.

:d) Returns: None

5) Class Method: `dispStorage`

:a) Definition: Display all the address-value pairs of the storage.

:b) Parameters: None

:c) Process: Use for loop to access all pairs.

:d) Returns: None

6) Static Method: `setVariable`

:a) Definition: The special storage connects to both the memory and register. It creates its data together with the data in the register or memory it is connected to.

:b) Parameters: `var`, `name`, `addr`, and `value`

:c) Process: Assign its data using the `name` as address and `addr` as value. Also, assign the data of `var` (either a memory or a register) using the `addr` as address and `value`.

:d) Returns: None

7) Static Method: `setVariables`

:a) Definition: Some storages in the register and memory have names with similar patterns. It initializes the special storage and the storage (register it is connected) with the same pattern.

:b) Parameters: `name`, `base`, and `stolen` (default value is zero)

:c) Process: Uses the `name` for the pattern and creates `stolen` (short for store length) of its data and the data in the register or memory ut us connected to. Its value (also equal to the register or memory address it is connected) starts from `base` and is incremented by 1 every loop.

:d) Returns: None

8) *Also Include*: The three storages (`memory`, `register`, and a special storage named `variable`) were created or instantiated. Several values were also created (see the explanation in the file below).

III. Addressing Modes

Filename: addressing.py This file contains a `Access` class for easy acces of memory and register and `AddressingMode` class.

Class: `Access`: Use to easily access and write the data in the memory and register.

1) Attributes: None.
2) Constructor: None
3) Static Method: `data`
   :a) Definition: Loads the value that follows the `flow` from the specified `address`.
   Example: `data('PC',['var','reg'])` means to start from 'PC' of the `variable` and gets its value. Then, use the value as address in the register. Finally, gets its value.
   :b) Parameters: `addr` and `flow`
   :c) Process: Start with the `addr` (it can be from the `variable`, `register`, or `memory`. Loop for every element of `flow` and make the current value as the address of the next storage based on `flow`.
   :d) Returns: The data of the storage specified on the last element of `flow`
4) Static Method: `store`
   :a) Definition: Store the `value` to the specified storage (memory or register) and address.
   :b) Parameters: `typ`, `addr` and `value`
   :c) Process: Gets the value based on the `addr` (address) and `typ` (either memory or register)
   :d) Returns: None

Class: `Addressing Mode`: The `Access` class is useful in this class.

1) Attributes: None.
2) Constructor: None
3) Static Method: `immediate`
   :a) Definition: Immediate addressing Mode
   :b) Parameters: `var`
   :c) Process: Convert `var` from Half Precision binary format to decimal
   :d) Returns: A value converted from Half Precision format.
4) Static Method: `relative`
   :a) Definition: Relative addressing mode with `displace`
   :b) Parameters: `displace`
   :c) Process: Gets the value of PC and add it to `displace`.
   :d) Returns: A value where the effective address is pointing.
5) Static Method: `based`
   :a) Definition: Based addressing mode with `displace`
   :b) Parameters: `displace`
   :c) Process: Gets the address of BR and add it to `displace`.
   :d) Returns: A value where the effective address is pointing.
6) Static Method: `indexed`
   :a) Definition: Indexed addressing mode with `displace`
   :b) Parameters: `displace`
   :c) Process: Gets the value of XR and add it to `displace`.
   :d) Returns: The effective address and the value it is pointing.
7) Static Method: `register`
   :a) Definition: Register addressing mode from the given address `reg_addr`
   :b) Parameters: `reg_addr`
   :c) Process: Converts `reg_addr` to decimal using Half Precision.
   :d) Returns: The effective address, the value, and the storage (register) it is pointing.
8) Static Method: `register_indirect`
   :a) Definition: Register indirect addressing mode from the given address `reg_addr`
   :b) Parameters: `reg_addr`
   :c) Process: Gets the value based on the address `reg_addr`.
   :d) Returns: The effective address and the value it is pointing.

9) Static Method: `direct`
   :a) Definition: Direct addressing mode from the given address `var_addr`
   :b) Parameters: `var_addr`
   :c) Process: Converts `var_addr` to decimal using Half Precision.
   :d) Returns: The effective address and the value it is pointing.
10) Static Method: `indirect`
   :a) Definition: Indirect addressing mode from the given address `var_addr`
   :b) Parameters: `var_addr`
   :c) Process: Gets the value based on the address `var_addr`.
   :d) Returns: The effective address and the value it is pointing.
11) Static Method: `autoinc`
   :a) Definition: Auto-increment addressing mode from the given address `reg_addr`
   :b) Parameters: `reg_addr`
   :c) Process: Gets the value based on the address `reg_addr`. Then, Increment the register value by 1 based on the address.
   :d) Returns: The effective address and the value it is pointing.
12) Static Method: `autodec`
   :a) Definition: Auto-decrement addressing mode from the given address `reg_addr`
   :b) Parameters: `reg_addr`
   :c) Process: Decrement the register value by 1 based on the address. Then, gets the value based on the address `reg_addr`.
   :d) Returns: The effective address and the value it is pointing.
13) Note: The immediate, based, and relative returns the value only because they are always on the second operand. Only the register mode returns the storage because the remaining modes (not mentioned) uses the memory as their storage.

IV. Instruction Code

This shows the format of the 32-bit Instruction code. It is used in converting commands into 32-bit Instruction code and vice versa

Instruction Code: Length: `32 bits`

OpCode					ib	Op1Mode			Op1Addr						
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

rb	Op2Mode			Op2Addr							Extra bits				
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31

1) OpCode (`Operation Code`): `5-bit` code that identifies the operations (ADD, JMP, MOV, etc.) located from `bit 0` to `bit 4` of the Instruction Code.

Operation	Execute	Write	Category Code	Definition
MOD	1	1	000	Remainder
ADD	1	1	001	Addition
CB	1	1	001	Create Block
CF	1	1	001	Create Function Block
SUB	1	1	010	Subtraction
CMP	1	1	010	Compare
MUL	1	1	011	Multiplication
DIV	1	1	100	Division
JEQ	1	0	000	Jump if equal
JNE	1	0	001	Jump if not equal
JLT	1	0	010	Jump if less than
JLE	1	0	011	Jump if less than or equal
JGT	1	0	100	Jump if greater than
JGE	1	0	101	Jump if greater than or equal
JMP	1	0	110	Jump
MOV	0	1	000	Move
ADDPC	0	1	000	ADDPC Relative
CALL	0	1	001	Call Function
RET	0	1	010	Return to caller
SCAN	0	1	011	Scan input (not required)
PRNT	0	0	000	Print input (not required)
EOP	0	0	001	End of main program
FUNC	0	0	001	Start of other function(s)

:a) EXECUTE `bit`: A `1-bit` code that identifies if the execute command (ADD, JMP, etc.) will be performed. It is located at the `bit 0` of the Instruction Code.

:b) WRITE `bit`: A `1-bit` code that identifies if the write command (ADD, MOV, etc.) will be performed. It is located at the `bit 1` of the Instruction Code.

:c) CATEGORY CODE: A `3-bit` code that differentiate operations with similar EXECUTE and WRITE `bits`. Located from `bit 2` to `bit 4` of the Instruction Code.

Here is the format of all operations:

Operation	Operand 1	Operand 2	Note
MOD	Yes	Yes	Op1 has no relative, based, and immediate
ADD	Yes	Yes	Op1 has no relative, based, and immediate
CB	Yes	None	Op1 is from 'B1' to 'B8' only
CF	Yes	None	Op1 is from 'F1' to 'F4' only
SUB	Yes	Yes	Op1 has no relative, based, and immediate
CMP	Yes	None	Computes Jump Register 'JR'
MUL	Yes	Yes	Op1 has no relative, based, and immediate
DIV	Yes	Yes	Op1 has no relative, based, and immediate
JEQ	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
JNE	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
JLT	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
JLE	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
JGT	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
JGE	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
JMP	Yes	None	Op1 is from 'B1' to 'B8' or 'F1' to F4'
MOV	Yes	Yes	Op1 has no relative, based, and immediate
ADDPC	Yes	Yes	Op1 has no relative, based, and immediate. Op2 is register, direct, or immediate
CALL	Yes	None	Op1 is from 'F1' to 'F4' only
RET	Yes	None	Op1 moves to 'ACC'
SCAN	Yes	Yes	Op2 is 'M:' concatenated by a message
PRNT	Yes	Yes	Op2 is 'M:' concatenated by a message
FUNC	None	None	End the Program

2) IB (IMMEDIATE `bit`): A `1-bit` code that identifies if the second operand has an `Immediate Addressing Mode`. Located at the `bit 5` of the Instruction Code. The `second operand` in this mode is located from `bit 17` to `bit 31` of the Instruction Code.

3) OP1MODE (`First Operand Addressing`): A `3-bit` code that identifies the `Addressing Mode` of the `First Operand`. Located from `bit 6` to `bit 8` of the Instruction Code.

:a) REGISTER ADDRESSING MODE: `000`

:b) REGISTER INDIRECT ADDRESSING MODE: `001`

:c) DIRECT ADDRESSING MODE: `010`

:d) INDIRECT ADDRESSING MODE: `011`

:e) INDEXED ADDRESSING MODE: `100` *with displacement from register or memory*

:f) INDEXED ADDRESSING MODE: `101` *with integer displacement*

:g) AUTO-INCREMENT ADDRESSING MODE: `110`

:h) AUTO-DECREMENT ADDRESSING MODE: `111`

4) OP1ADDR `First Operand Address`: A `7-bit` code that identifies the `address` of the `First Operand`. Located from `bit 9` to `bit 15` of the Instruction Code.
In the INDEXED ADDRESSING MODE, the leftmost signifies the displacement type.

:a) *displacement from the register*: `0`

:b) *displacement from the memory*: `1`

:c) *positive integer displacement*: `0`

:d) *negative integer displacement*: `1`

Note: *the displacement has 6 bits only so the maximum value or address is 63, so be careful on choosing the displacement*

5) RB (RELATIVE `bit`): A `1-bit` code that identifies if the second operand has a `Relative or Based Addressing Mode` if and only if the IMMEDIATE `bit` is off (zero). Located at the `bit 16` of the Instruction Code.

6) OP2MODE (`Second Operand Addressing`): A `3-bit` code that identifies the `Addressing Mode` of the `Second Operand`. Located from `bit 17` to `bit 10` of the Instruction Code.

:a) REGISTER ADDRESSING MODE: `000`

:b) Register Indirect Addressing Mode: `001`
:c) Direct Addressing Mode: `010`
:d) Indirect Addressing Mode: `011`
:e) Indexed Addressing Mode: `100` *with displacement from register or memory*
:f) Indexed Addressing Mode: `101` *with integer displacement*
:g) Auto-Increment Addressing Mode: `110`
:h) Auto-Decrement Addressing Mode: `111`

The following Addressing Modes are allowed on the second operand only. The Relative `bit` must be `1` but the Immediate `bit` must be `0`.

:a) Based Addressing Mode: `000` *with displacement from register*
:b) Based Addressing Mode: `001` *with displacement from memory*
:c) Based Addressing Mode: `010` *with positive displacement*
:d) Based Addressing Mode: `011` *with negative displacement*
:e) Relative Addressing Mode: `100` *with displacement from register*
:f) Relative Addressing Mode: `101` *with displacement from memory*
:g) Relative Addressing Mode: `110` *with positive displacement*
:h) Relative Addressing Mode: `111` *with negative displacement*

7) Op2Addr `Second Operand Address`: A `7-bit` code that identifies the `address` of the `Second Operand`. Located from `bit 20` to `bit 26` of the Instruction Code.
In the Indexed Addressing Mode, the leftmost signifies the displacement type.
:a) *displacement from the register*: `0`
:b) *displacement from the memory*: `1`
:c) *positive integer displacement*: `0`
:d) *negative integer displacement*: `1`
Note: *the displacement has 6 bits only so the maximum value or address is 63, so be careful on choosing the displacement*

8) `Extra bits`: so far, the extra bits are used for the Immediate Addressing Mode of the `second operand`. Located from `bit 27` to `bit 31` of the Instruction Code.

V. Compile the Instructions

Filename: compiler.py This file contains the `Instruction` class for converting every instruction to binary Instruction Code. It also contains two global variables for converting operation into OpCode.

`operations`: Contains different operations grouped by their Execute and Write `bits`.

`operationCodes`: Contains 2 groups. The first group is the Execute and Write `bits`. The second group is the Category Code.

Note: Each group in the `operations` represents the first group in the `operationCodes`. Each member of the group in the `operations` represents the second group in the `operationCodes`.

Class: `Instruction`: Converts the instructions of the program into Instruction Codes.

1) Attributes: None.
2) Constructor: None
3) Static Method: `decodeMSG`
:a) Definition: Coverts each dash to space, underscore to tab, dash+underscore to newline. Also converts the word minus to dash, and the word under to underscore.
:b) Parameters: `msg`
:c) Process: Replace all characters or groups of characters into specified characters defined above.
:d) Returns: The new `msg` where specified values were replaced.
4) Static Method: `encodeOp`
:a) Definition: Convert the `operand` into Operand Code.
:b) Parameters: `operand`
:c) Process: If the `operand` is a number (float or integer) – meaning immediate, return the Half Precision binary format of the operand. If the `operand` contains 'M:' – meaning a message, store the message to appropriate storage (this is not related to `ISA` so it is optional. Otherwise do the following:
α·) If the operand contains open and closed parenthesis (remove them), identify whether the addressing mode is relative (contains 'Z'), based (contains 'Y'), or indexed (contains 'X') because the three have a common format (remove 'X', 'Y', or 'Z'). What remains is either an integer, register address, or memory address. Create the appropriate Addressing Mode binary code and concatenate it with binary conversion (if integer) or the address (if register or memory) of the remaining string. Return the resulting string.

$\beta\cdot$) Still under operand that contains the open and close parenthesis. If the remaining string contains '+' (auto-inc), '-' (auto-dec), 'R' − including 'PC' and 'ACC' (register indirect), otherwise (indirect), create the appropriate Addressing Mode binary code and remove ('+' or '-' for auto-inc/dec).

$\gamma\cdot$) Otherwise (the operand doesn't have any parenthesis), identify if it contains 'R' − including 'PC' and 'ACC' (register), otherwise (direct). Then, create the appropriate Addressing Mode binary code.

$\delta\cdot$) What remains are the auto-inc/dec, register (direct or indirect), or memory (direct or indirect). Concatenate the Addressing Mode binary code to the address of the remaining string. Note: *auto-inc/dec are similar to register indirect, so the remaining string is a register. Return the concatenated Addressing Mode binary code and the address of the operand.*

:d) Returns: The Half Precision of the operand if immediate, Otherwise, the concatenation of the Addressing Mode binary code and the address or binary conversion of the operand. Note: *always make sure that the length of the binary code is 10, fill it with leading zeros if not using the* `zfill()`.

5) Static Method: `encode`

:a) Definition: Encode the instruction into Instruction Code.

:b) Parameters: `inst`

:c) Process: If the operation is 'FUNC' return an instruction code of all zeros. Be sure to use `zfill()` or `addZeros()` to fix its length. Otherwise, do the following:

$\alpha\cdot$) There are operations with similar OpCode. You have to convert them to the basic operation. There are also operations with implicit actions so, you have to complete them.

Operation	E	W	Cat	Simplify to
`CB`	1	1	001	Add 'BR' to Block
`CF`	1	1	001	Add 'BR' to Function Block
`CMP`	1	1	010	Subtract operand from 'JR'
`J__`	1	0	110	Compare 'JR' to zero based on condition
`ADDPC`	0	1	000	Move relative address to operand
`CALL`	0	1	001	Move 'PC' to 'CR', then move Function Block to 'PC'
`RET`	0	1	010	Move 'CR' to 'PC', then move operand to 'ACC'
`FUNC`	0	0	001	Same as `EOP`

Then, create the OpCode of the new Instruction.

$\beta\cdot$) Concatenate the Opcode to the encoded first operand. If there is a second operand, concatenate the encoded second operand to the Opcode and encoded first operand.

:d) Returns: The `32-bit` Instruction Code.

6) Static Method: `encodeProgram`

:a) Definition: Encode each instruction and put them to 'PC' in order except for 'CF' and 'CB'.

:b) Parameters: `program` − this set of instructions comes from a file.

:c) Process: Usually, instructions are in order except when there is a jump. When the created block was not visited/executed, we need to skip several instructions just to go to that instruction. Because of that, the flow will be too messy. With that, we will move all the 'CB' and 'CF' instructions at the start of the program. But we'll discuss it later.

$\alpha\cdot$) Start with initializing the address of the first instruction (address of 'BR'), a list of instructions (we will insert the 'CB' and 'CF' at the start), counter for blocks (to know where will we insert the instruction with 'CB' or 'CF'), and a variable for multiline comment (start as false).

$\beta\cdot$) We will loop for every instruction of the `program`. The first is to skip the comments. We have 'x' for single-line comments and 'z' for multiline comments. We will first skip lines with empty instructions (or only have space and or tab − though this is optional). Next is to check if the first character is 'z', then the following instructions will be considered comments until we encounter another 'z'. Lastly, we will check if the first character is an 'x' or the variable for the multiline comment is set to true.

$\gamma\cdot$) Next is to store the message if the instruction has a message (this is optional). Then, if the operation of the instruction is 'CB' or 'CF', we will store the current address in the block register operand. Make sure that it is in Half Precision format (convert it if not). Then, insert the encoded instruction into the ith element (i is based on the block counter) of our list. Increment the block counter.

$\delta\cdot$) If the operation of the instruction is neither 'CB' nor 'CF', append the encoded instruction at the last of the list.

$\epsilon\cdot$) This is now outside the loop. We will store the number of blocks to 'BR', this will play a very important role in computing where to find the actual block in the instruction.

$\zeta\cdot$) Finally, put the encoded instruction (Instruction Code of the list to the memory starting from the address of 'BR'. Don't forget to increment for the next address or to make sure that your for loop automatically increments it.

:d) Returns: None

VI. Running the Program

Filename: run.py
This file contains two classes. The EXCEPT class handles the exception while the PROGRAM class runs the Instruction Codes from the memory pointed by 'PC'.

CLASS: `Except`:

1) ATTRIBUTES: The message for an exception and the occurrence of the exception.
   :a) `message`: The message of the exception (usually used when there's an exception)
   :b) `occur`: Tells whether the exception occurs or not.
   :c) `ret`: Return value of the exception. Different from the message.
2) CONSTRUCTOR: Initialize the `message` and `occur`
   :a) PARAMETERS: `msg` and `occur` (default value is TRUE).
   :b) PROCESS: Set the `message` and the `occur`.
3) CLASS METHOD: `dispMSG`
   :a) PARAMETERS: None
   :b) PROCESS: Print the `message`
   :c) RETURNS: None
4) CLASS METHOD: `isOccur`
   :a) PARAMETERS: None
   :b) PROCESS: Return TRUE if `occur` is TRUE, Otherwise, FALSE.
   :c) RETURNS: The `occur`
5) CLASS METHOD: `setReturn`
   :a) PARAMETERS: `value`
   :b) PROCESS: Set the value of `ret`
   :c) RETURNS: None
6) CLASS METHOD: `getReturn`
   :a) PARAMETERS: None
   :b) PROCESS: None
   :c) RETURNS: The return value of the exception

CLASS: `Program`:

1) CONSTRUCTOR: Encode the `program`
   :a) PARAMETERS: `program`
   :b) PROCESS: Encode each Instruction of the program.
2) STATIC METHOD: `exception`
   :a) DEFINITION: Finds the exception based on its `name` and `value`.
   :b) PARAMETERS: `name` and `value`
   :c) PROCESS: It is an IF... ELIF... ELSE (else is optional) statement. Though, the 'DivByZero' currently exists. The return value is 'Infinity' if both the operands are zero, and 'undefined' if only the operand 2 is zero.
   :d) RETURNS: The exception.
3) CLASS METHOD: `write`
   :a) DEFINITION: Perform Write operations.
   :b) PARAMETERS: `dest`, `src`, and `movecode`
   :c) PROCESS: If `movecode` is `1` − means a 'CALL' operation, you have to move the value of 'PC' to 'CR'. If the `movecode` is `2` − means a 'RET' operation, you have to move the value of 'CR' to 'PC'. If the `movecode` is `3` − means a 'SCAN' operation, you change the `src` by the value of the message.
   Whatever is the value of the `movecode`, the default move operation (`src` to `dest`) must be performed.
   :d) RETURNS: None
4) CLASS METHOD: `execute`
   :a) DEFINITION: Perform Execute operations.
   :b) PARAMETERS: `result` and `opcode`
   :c) PROCESS: Get the category from the `opcode`.
      $\alpha\cdot$) If the WRITE BIT of the `opcode` is 1, perform the four basic operations and modulo based on the category and return the result. For division, create an exception for division by zero. Return the return value of the exception if division by zero occurs.

$\beta\cdot$) Otherwise (WRITE BIT of the `opcode` is 0), perform jumps based on the category. Note: The purpose of jump with comparison is really to compare the 'JR' and zero. Meaning we need the Difference because $x >= y$ also means $x - y >= 0$. That's the reason why the 'CMP' operation just stores the difference to 'JR'.

:d) RETURNS: The result if WRITE BIT of the `opcode` is 1.

5) CLASS METHOD: `getOp`
   :a) DEFINITION: Gets the effective address and the type of storage (memory or register) of the operand.
   :b) PARAMETERS: `inscode` (the code of operand including addressing mode)
   :c) PROCESS: Identify the addressing mode.
      $\alpha\cdot$) If the addressing mode is immediate, return its Half Precision decimal value. If it is based, indexed, or relative, Identify the displacement. Otherwise, identify the storage (memory or register) address (make sure to get the Half Precision decimal value).
      $\beta\cdot$) Take note of the parameters of all the addressing modes identified (except for immediate). Call the appropriate addressing mode with the appropriate parameter.
   :d) RETURNS: The addressing mode except for immediate.
6) CLASS METHOD: `run`
   :a) DEFINITION: Execute each INSTRUCTION CODES starting from the address pointed by 'IR'
   :b) PARAMETERS: None
   :c) PROCESS: Initialize a list of monadic and niladic operations. They are for future use but for now, it's empty because all operations are converted to instructions with two operands. Create a loop that the condition is always TRUE
      $\alpha\cdot$) Gets the value of 'IR'. Get the INSTRUCTION CODE pointed by the 'IR'. If the INSTRUCTION CODE is not a `32-bit` format or consists of all zeros, break from the loop.
      $\beta\cdot$) Get the `opcode`, the first operand, and the second operand (only if the operation is not dyadic).
      $\gamma\cdot$) If the EXECUTE BIT is 1, perform the `execute` with appropriate parameters. If the WRITE BIT is 1, perform the `write` with appropriate parameters.
      $\delta\cdot$) If both the EXECUTE and WRITE BIT are all zeros, perform the print.
      $\epsilon\cdot$) Move the value of 'PC' to 'IR', then increment the value of 'PC' by 1.
   :d) RETURNS: None
7) *Also Included*: Running the instructions from a file. Create a variable for division by zero exception. Access the file (it must have an extension with the shortcut of your group name). Convert the text from the file as a list of instructions and pass it to the `Program` class. Finally, the `Program` class calls the `run`.

VII. Distribution of Task

The `bin_convert.py` and `storage.py` will be given. The `data` method of the `Access` class will be also given. There will be 15 students working on the remaining part of the project.

1) The `store` method of `Access` class from `addressing.py`. Commenting part of `encodeProgram` method of `Instruction` class from `compiler.py`.
2) The `immediate`, `relative`, and `based` methods of `AddressingMode` class from `addressing.py`. Reading the file with instructions and running the `Program` from the `run.py`.
3) The remaining methods of `AddressingMode` class from `addressing.py`.
4) Converting some instructions (`ADDPC`, `CALL`, `RET`, `CB`, `CF`, `CMP`, `FUNC`, and all the jump commands) into another instruction in the `encode` method of class `Instruction` from `compiler.py`.
5) Convert the instruction into instruction code in the `encode` method of class `Instruction` from `compiler.py`.
6) Convert the operand to immediate, auto-increment/decrement, register, register indirect, direct, and indirect in the `encodeOp` method of class `Instruction` from `compiler.py`.
7) Convert the operand to displacement (relative, based, and indexed) in the `encodeOp` method of class `Instruction` from `compiler.py`.
8) The `encodeProgram` method of class `Instruction` from `compiler.py`.
9) The constructor and the `exception` method of the `Program` class and the `Except` class from `run.py`.
10) Looping and getting the `OpCode` and the operands as well as updating the 'IR' and 'PC' in the `run` method of the `Program` class from `run.py`.
11) The execute, write, and no-execute-no-write part in the `run` method of the `Program` class from `run.py`.
12) The displacement (relative, indexed, and based) part in the `getOp` method of the `Program` class from `run.py`.
13) The other (direct/indirect, register direct/indirect, auto-increment/decrement) part in the `getOp` method of the `Program` class from `run.py`.
14) The `execute` method of the `Program` class from `run.py`.
15) The `write` method of the `Program` class from `run.py`.