Create a Python code for this. Both the Binary Conversion and Storage will be given.

I. Conversion

Filename: 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 `Precision` class.
   :b) `fraction`: The length of the f for the `Precision` class.
   :c) `precision`: The length of the `Precision` format.
   :d) `instrxn`: The length of the our `ISA Instruction` format.
   :e) `dec_place`: The default length of decimal places the `Precision` will be rounded off to.
   :f) `opAddr`: The length of the operand address.
   :g) `opMode`: The length of the addressing mode.
   :h) `operand`: The length of the operand.
2) Constructor: None
3) Static Method: `trimDec`: Rounds off a string into `dec_place` decimal places.
4) Static Method: `addZeros`: Fill zeros to the left or the right of the `value` based on the length specified by `strlen`.

Class: `Value`: For checking the existence or the type of a value.

1) Attributes: None
2) Constructor: None
3) Static Method: `isNumber`: Check if a string `value` is a number.
4) Static Method: `isInteger`: Check if a string `value` is an integer.
5) Static Method: `inRegister`: Check if `value` is a register.

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

1) Attributes: None
2) Constructor: None
3) Static Method: `idec2bin`: Converts a fractional decimal to a fractional binary.
4) Static Method: `ibin2dec`: Converts a fractional binary to a fractional decimal.

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

1) Attributes: None
2) Constructor: None
3) Static Method: `spbin2dec`: Converts a Single Precision binary format to a decimal.
4) Static Method: `dec2spbin`: Converts a decimal to a Single Precision binary format.
5) Static Method: `spbin2bin`: Converts a Single Precision binary format to a binary.
6) Static Method: `bin2hpbin`: Converts a binary to a Single Precision binary format.
7) *Also Included*: A tester on how many decimal values are correctly converted by the 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`
3) Class Method: `load`: Loads the value on the specified `address`.
4) Class Method: `store`: Store or create the `value` on the `data` given the `address`.
5) Class Method: `setStorage`: set the value of all the empty slots to zero.
6) Class Method: `dispStorage`: Display all the address-value pairs of the storage.
7) Class Method: `dispStorageSlot`: Display the storage slot based on the address.

8) Static Method: `setVariable`: 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.
9) Static Method: `setVariables`: Some storage 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.
10) Static Method: `setTmpVariable`: Define variable *(for future use)*.
11) Static Method: `setTmpVariables`: Define a list of variables *(for future use)*.
12) Static Method: `removeVariables`: Remove all the defined variables *(for future use)*.
13) Static Method: `removeVariable`: Remove a defined variable *(for future use)*.
14) *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 at the bottom part of the file). Some important registers are:
    :a) `R#` (`General Purpose Registers`): From R1 to R7.
    :b) `BR` (`Base Register`): The address of the first instruction in the program.
    :c) `IR` (`Instruction Register`): The address of the current instruction of the program.
    :d) `PC` (`Program Counter`): The address of the next instruction of the program.
    :e) `SPR` (`Stack Pointer Register`): The address of the stack bottom.
    :f) `TSP` (`Top Stack Pointer`): The address of the stack top.
    :g) `BPR` (`Block Pointer Register`): The address of the first block.
    :h) `NBP` (`Next Block Pointer`): The address of the next block to be defined.
    :i) `MPR` (`Message Pointer Register`): The address of the first message.
    :j) `NMP` (`Next Message Pointer`): The address of the next message to be filled.
    :k) `I1` (`Array Index 1`): The address that contains the index of the first operand (indexed addressing only).
    :l) `I2` (`Array Index 2`): The address that contains the index of the second operand (indexed addressing only).
    :m) `A#` (`Array Pointers`): Contains the address of an array's first element. From A1 to A4.

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`: Loads the value that follows the `flow` from the specified `address`.
4) Static Method: `store`: Store the `value` to the specified storage (memory or register) and address.

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

1) Attributes: None.
2) Constructor: None
3) Static Method: `immediate`: Immediate addressing Mode *(not included)*.
4) Static Method: `relative`: Relative addressing mode with `displace` *(not included)*.
5) Static Method: `based`: Based addressing mode with `displace` *(not included)*.
6) Static Method: `indexed`: Indexed addressing mode with `displace`.
7) Static Method: `register`: Register addressing mode from the given address `reg_addr`.
8) Static Method: `register_indirect`: Register indirect addressing mode from the given address `reg_addr`.
9) Static Method: `direct`: Direct addressing mode from the given address `var_addr`.
10) Static Method: `indirect`: Indirect addressing mode from the given address `var_addr`.
11) Static Method: `autoinc`: Auto-increment addressing mode from the given address `reg_addr`.
12) Static Method: `autodec`: Auto-decrement addressing mode from the given address `reg_addr`.
13) Static Method: `stack`: Stack addressing mode with stack option (push, pop, or top). Pop and top returns the address of the stack top. Push inserts the address at the stack top while pop removes the address from the stack.

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

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.

Opn	E	W	Cat	Opd	Definition
PRNT	0	0	000	2	Print message from first operand address to second operand address (not required).
PRNT	0	0	000	1	Print value of operand.
EOP	0	0	001	2	Print message and End of main program (not required).
EOP	0	0	001	1	Print operand and End of main program.
MOV	0	1	000	2	Move the second operand to first operand.
PUSH	0	1	001	1/2	Push the operand(s) to the stack.
POP	0	1	010	1/2	Pop the stack and move it to the operand(s).
CALL	0	1	011	1	Push the address of PC to the stack. Move the address of the first operand to the PC.
RET	0	1	100	1	Pop the stack and move the value to the PC. Push the address of the first operand to the stack.
SCAN	0	1	101	1	Scan the input and move it to the first operand.
DEF	0	1	110	1	Create a Function during compile whose value is the address of the next instruction.
DEB	0	1	110	1	Create a Block during compile whose value is the address of the next instruction.
DEV	0	1	110	1	Create a variable (not required).
For all conditional jumps					Use SUB Op2 Op3 and change the instruction to J-- Op1 Op2
JEQ	1	0	000	3-2	Jump to the first operand if the two operands are equal.
JNE	1	0	001	3-2	Jump to the first operand if the two operands are not equal.
JLT	1	0	010	3-2	Jump to the first operand if the first operand is less than the second operand.
JLE	1	0	011	3-2	Jump to the first operand if the first operand is less than or equal to the second operand.
JGT	1	0	100	3-2	Jump to the first operand if the first operand is greater than the second operand.
JGE	1	0	101	3-2	Jump to the first operand if the first operand is greater than or equal to the second operand
JMP	1	0	110	1	Jump to the first operand.
MOD	1	1	000	2	Remainder of first operand divided by the second operand.
ADD	1	1	001	2	Sum of the two operands.
SUB	1	1	010	2	Difference of the first operand subtracted by the second operand.
MUL	1	1	011	2	Product of the 2 operands.
DIV	1	1	100	2	Quotient of the first operand divided by the second operand.

:a) Operation (Opn): Type of operation.

:b) Execute (E): 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.

:c) Write (W): 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.

:d) Category Code (car): A 3-bit code that differentiate operations with similar Execute and Write bits. Located from bit 2 to bit 4 of the Instruction Code.

2) Op1Mode (First Operand Addressing): A 3-bit code that identifies the Addressing Mode of the First Operand. Located from bit 5 to bit 7 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 index regiter 1 or I1*

:f) Stack Addressing Mode: 101 *for* Push

:g) Stack Addressing Mode: 110 *for* Pop

3) Op1Addr First Operand Address: A 8-bit code that identifies the address of the First Operand. Located from bit 8 to bit 15 of the Instruction Code.

4) Op2Mode (Second Operand Addressing): A 3-bit code that identifies the Addressing Mode of the Second Operand. Located from bit 16 to bit 18 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 index regiter 2 or I2*

:f) Stack Addressing Mode: 101 *for* Push

:g) Stack Addressing Mode: 110 *for* Pop

5) Op2Addr Second Operand Address: A 8-bit code that identifies the address of the Second Operand. Located from bit 19 to bit 26 of the Instruction Code.

6) Extra bits: Located from bit 27 to bit 31 of the Instruction Code. *(non-functional).*

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 group member 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`: Converts the 32-bit data into 5 characters *(no required)*.
4) Static Method: `preEncode`: Some instruction needs to be adjusted or converted to two instructions, such as 'DEF', 'DEV'. conditional jumps, and indexing. This adjustment or conversion is done using this method.
5) Static Method: `encodeOp`: Convert the `operand` into 8-bit Operand Code.
6) Static Method: `encode`: Encode the instruction into 32-bit Instruction Code.
7) Static Method: `encodeProgram`: Encode each instruction and put them to 'PC' in order, except for 'DEF' and 'DEB'.

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`
3) Class Method: `dispMSG`: Print the `message`.
4) Class Method: `isOccur`: Return True if `occur` is True, Otherwise, False.
5) Class Method: `setReturn`: Set the value of `ret`.
6) Class Method: `getReturn`: Returns the value of the exception

Class: `Program`:

1) Constructor: Encode the `program`
   :a) Parameters: `program`
   :b) Process: Pre-encode and encode each Instruction of the program.
2) Static Method: `exception`: Finds the exception based on its `name` and `value`.
3) Class Method: `write`: Perform Write operations.
4) Class Method: `execute`: Perform Execute operations.
5) Class Method: `getOp`: Gets the effective address and the type of storage (memory or register) of the operand.
6) Class Method: `run`: Execute each Instruction Codes starting from the address pointed by 'IR'.
7) *Also Included*: Running the instructions from a file. Create a variable for the 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 `convert.py` and `storage.py` will be given. Other files will some code will also be given. You can add new functions, but do not change or remove what was given.

1) For a team with 4 members:
   :a) One member for the `addressing.py`.
   :b) One members for the `compiler.py`.
   :c) Two members for the `run.py`. One member for functions `getOp` and `run`, the other member will do the rest.

2) For a team with 3 members:
   :a) One member for the `addressing.py`, the `Except Class` and the `exception` method of `Program Class` of `run.py`.
   :b) One members for the `compiler.py` and the running part (or the 'also included' section) of `run.py`.
   :c) Two members for the `run.py`.