Design document for the ML

I. Instruction Syntax:

OPCODE TYPE OPERAND1 OPERAND2

- **Explanation:**

The first two characters in an instruction are a sign bit (+ or -) and a digit (0 to 9). The combinations that can be made out of these two characters build the opcode. Please refer to page 2 (ML instruction set) to know the opcode of each instruction.

Our memory has 20000 words, 10000 are allocated for data memory, while 10000 are for code memory. This means that each operand takes four digits (0-9999).

The one digit left in the instruction refers to the type, it comes before the two operands, and it is used to describe the type of each operand (Positive literal, negative literal, address). Here is a table that maps between the type-digit and the two operands:

Type Digit	Operand 1	Operand 2
1	N	N
2	N	P
3	N	A
4	P	N
5	P	P
6	P	A
7	A	N
8	A	P
9	A	A

(P: Positive literal; N: Negative Literal; A: Address)

e.g.,

+0 5 0029 3814

- ⇨ +0 is the opcode for assignment.
- ⇨ The 5 right after the opcode tells us that the first operand is a positive literal (+29) and the second operand is an address.
- ⇨ Meaning: assign the value 29 to the memory location 3818.

II. ML Instruction Set:

Opcode	Meaning	Instruction Example	Explanation	Comment
+0	Assign	+0 6 0001 0000	Assign the positive literal 1 to the memory address 0000	The only possible types of the operands are: 3 : N A 6 : P A 9 : A A
		+0 3 0212 5362	Assign the negative literal -212 to the memory address 5362	
		+0 9 4382 3292	Assign the value at the memory address 4382 to the memory address 3292	
-0	Move value from accumulator	-0 0 0000 3294	Move the value in the accumulator to the memory location 3294	The second operand is always a memory location. The type-digit used here is 0, which is not used in any other case.
+1	Increment	+1 6 0001 4832	Increment by the value 1 the value in memory address 4832	The operands can be any of the possible types.
-1	Decrement	-1 6 0002 4832	Decrement by the value 2 the value in memory address 4832	
+2	Add	+2 4 0192 0007	Add the positive literal 192 to the negative literal -7 and store the result in the accumulator.	The operands can be of any of the possible types (negative/positive literal, address). Since an additional operand is needed, it is always assumed to be the accumulator.
-2	Subtract	-2 4 0192 0007	Subtract the positive literal 192 from the negative literal -7 and store the result in the accumulator.	
+3	Multiply	+3 4 0192 0007	Multiply the positive literal 192 by the negative literal -7 and	

			store the result in the accumulator.	
-3	Divide	-3 4 0192 0007	Divide the positive literal 192 by the negative literal -7 and store the result in the accumulator.	
+4	If equal	+4 7 8324 0003	If the value at memory address 8324 is equal to the negative literal -3, execute the jump statement in the next line.	The operands can be of any of the possible types.
-4	If not equal	-4 7 8324 0003	If the value at memory address 8324 is not equal to the negative literal -3, execute the jump statement in the next line.	
+5	If greater than	+5 7 8324 0003	If the value at memory address 8324 is greater than the negative literal -3, execute the jump statement in the next line.	
-5	If less than	-5 7 8324 0003	If the value at memory address 8324 is less than to the negative literal -3, execute the jump statement in the next line.	
+6	Jump to	+6 0 0000 0008	Jump to code memory address location 0008.	The first operand is not used. The second operand can only be a code memory address.
+7	Read	+7 0 0000 3277	Read an input value and store it in memory address 3277.	The first operand is not used. The second operand can only be a data memory address.
-7	Print	-7 9 3728 0000	Print the value stored in memory address 3728.	The first operand can only be a data memory address.
+9	Stop	+9 9 9999 9999 +9 0 0000 0000	Separator Halt	

III. Interpreter:

The interpreter takes as input the output file of the assembler (ML Code.txt) and follows the decode-execute cycle to execute the code. It starts off by reading the data initialization part and storing the data into the data memory array that is used to simulate data memory. Then, it reads the program part into the code memory array.

Next, the interpreter starts the decode-execute cycle, it loops over the program, and extracts the opcodes. If the current instruction is not a halt, it keeps fetching the instructions. For each instruction, the operands are being fetched and then the instruction is executed.

We have designed the expected input files of the interpreter, in addition to comments and expected outputs after running the interpreter.