Design document for the AL

I. Instruction Syntax:

OPCODE TYPE OPERAND1 OPERAND2

- **Explanation:**

The opcode is a symbol that can take up to three characters.

Since an operand can be an address, a positive literal value, or a negative literal value, we used one digit of the instruction to describe the operands. It uses following mapping:

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)

The two operands can take up to four characters. An operand can be a symbol referring to a data memory address. It can also be a label referring to a code memory address, or a code memory address itself, but this can only occur for the second operand. Operands can also be positive or negative literal values ranging from -9999 to 9999.

II. AL Instruction Set :

The following instruction set describes the instructions used by this assembly language and their corresponding opcode in machine language.

Assembly Code Symbol	AL Instruction Example	Meaning	ML Opcode
ASN	ASN 3 0212 5362 ASN 6 0000 7877 ASN 9 0000 0000	*Assign*: Assign a negative literal or positive literal or a value of a memory location to a memory location.	+0
MOV	MOV 0 0000 3294	*Move*: Move value from the accumulator to a memory address	-0
INC	INC 6 0001 3123	*Increment:* Increment the second operand by the value specified in the first operand	+1
DCR	DCR 6 0003 3100	*Decrement:* Decrement the second operand by the value specified in the first operand	-1
ADD	ADD 4 0192 VAR1	*Add:* Add the two operands and store the result in the accumulator	+2
SUB	SUB 6 3821 9384	*Subtract:* Subtract the two operands and store the result in the accumulator	-2
MUL	MUL 6 3821 9384	*Multiply:* Multiply the two operands and store the result in the accumulator	+3
DIV	DIV 6 3821 9384	*Divide*: Divide the two operands and store the result in the accumulator	-3
EQL	EQL 7 4635 0062	*If Equal:* If the first operand is equal to the second	+4

		operand, execute the jump statement in the next line. Otherwise skip the jump and continue.	
NEQ	NEQ 7 4635 0062	*If Not Equal:* If the first operand is not equal to the second operand, execute the jump statement in the next line. Otherwise skip the jump and continue.	-4
IGT	IGT 7 4635 0062	*If greater than:* If the first operand is greater than the second operand, execute the jump statement in the next line. Otherwise skip the jump and continue.	+5
ILT	ILT 7 4635 0062	*If less than:* If the first operand is less than the second operand, execute the jump statement in the next line. Otherwise skip the jump and continue.	-5
JMP	JMP 0 0000 LOOP JMP 0 0000 9434	*Jump to:* Jump to the code memory location specified in the second operand.	+6
INP	INP 0 0000 7382	*Read Input:* Read an input and store it in the data memory location specified by the second operand.	+7
OUT	OUT 0 8938 0000	*Print Output:* Print the value in the memory location specified by the first operand.	-7
SPR/HLT	SPR/HLT	*separator*/*halt*	+9
DEC	DEC VAR1 0004	*Declare:* Declare variable VAR1 that has the size specified in the second operand	

LBL	LBL 0 LOOP 0000		

III. Assembler and assembly language samples:

We have designed three program samples written in assembly language following the rules stated above. The simple code is used to do simple arithmetic operations (+, -, /, *) on two input values; in case of division by zero the division is not handled. The medium complexity code calculates the power (A to the power of B) of two input values using a loop; it handles the case where the exponent is zero in which the power is 1. The high complexity code is a program that prints a pattern depending on the number of rows entered by the user, for example if the number of rows is 4, the program displays the pattern 4 4 4 4 3 3 3 2 2 1; in case the number of rows is 0, the number 0 should be displays; and in case the number of rows is negative, the program ends. We have also designed the expected output files of the assembler for each of these samples.

The assembler takes as input an AL program file. The input file to our assembler is by default the simple code, it is defined as a macro; if you would like to run the assembler using another input file, you can copy the file name from a comment written right under the pre-processor directives.

The assembler starts off by reading the data initialization part from the input file and writing it at the same time in a 2D array that stores the machine language code. While doing this, the symbol table, which is represented by a 3D array of two string columns, is being built. Once reaching the separator and writing it to the code array, the assembler starts reading the program part and writing it to the code array. At the same time, the label table is being built. All the labeling instructions are not written to the code array because our machine language code should not have labels in it. If the second operand of a jump instruction happens to be a label, it is kept as is because its corresponding label entry might not be in the label table yet. Once reading the whole program part (reaching the separator), the labels are replaced by their corresponding code memory addresses using the label table. Then the assembler starts reading the input data to the code array. After reaching the end of the input file, the data is copied from the code array to an output file named ML Code.txt.