Introduction:

The Pullet 16 Assembler program is a simulation of a how a program would convert programs written in assembly language opcodes into machine language, that the CPU will then be able to execute its set instructions. The Pullet Machine will deal with 4096 words that will each be 16 bit each. The user will also be able to access all memory. This Pullet 16 Assembler program will be a 2-pass Assembler. This program will have seven classes, consisting of a Codeline class that will act as a container for individual lines of code, a Namespace class that will keep track of all global constants, a Hex class that will deal with hex operands, a Main class that will run through the program, a OneWordMemory class that acts as a container for one word of memory, a Pullet16Assembler class that will assemble the code, and a Symbol class taking act as a container for symbols. Each class will be described throughout this manual.

Header files:

Each of these classes with have its accompanying header file. In each of the header files the global variables for each individual class will be created as well as any data structures that are used.

Codeline Class:

This class will be used as a container to store lines of code that will then be decoded into different categories. Values will be set once the lines of code have been set to their appropriate category. These categories will consist of the codes being stored as a line counter, program counter, label, mnemonic, address, symbolic operand, hex operand, and comments. The line counter will keep track of what line is being read from the source code, and advance the counter as needed till the end of the source file. As lines are taken from the source file to be analyzed,

the counter will be printed out to the user and is used as a tool to check if errors occur while reading the source file line by line. The program counter will be much of the same, it will just keep track of the register in this simulated computer processor that will contain the memory address of the current instruction that is being executed. These addresses will also be printed out to the user and can be used for error checking results. The label function will check to see if any labels are present in the source file. The mnemonic function will take the opcode and convert it to its corresponding mnemonic. For example, if we look at the Machine code of an instruction, a 16-bit number, the first four numbers will be the opcode instruction. 1010 would represent an opcode for the mnemonic for Load, LD. The address function will take care of the asterisk found in the source file and determine whether it is indirect of indirect referencing. Direct referencing happens when the address specifies which word or register contains the operand. Indirect referencing happens when the word or register does not contain the operand itself, but the address of the operand. The symbolic operand function will come into play with a symbol table. The symbols, IDX, N, NN, will act like an opcode telling the CPU where the location of that symbol is. The symbols are just opcodes, and not the actual symbols themselves that will be found in the source file. The Symbol class will also come into play with this function (see Symbol Class). The hex operand function will act much the same as the symbolic, except it will look for the hex numbers in the file. The Hex class will be used in determining if the hex values are legal (see Hex Class). The comment function is the last one in this class. This function will look through the source file just for comments and filter those out. Those lines will then be set in the output with a comment opcode. Comments do not have anything to do with the assembly process but need to be checked so they do not interfere with any other codes.

Namespace Class:

This class will be used to declare global constants that will be used throughout in all other classes. Functions BitStringtoDec, DecToBitString, and GetMnemonicFromBits are declared. BitStringtoDec will convert a bit string to decimal. DecToBitString will do the opposite. Get MnemonicFromBits will take the bit string code and convert to text mnemonic.

Hex Class:

We make this class, because we need to run a series of tests to make sure the operands are legal. The functions is_invalid_, is_negative_, is_null, value_, error_messages_, and text_. Is_invalid_ will check to see if the operand that is detected is invalid or not. If HasAnError() is true, then is_invalid_ will also be true. Is_negative_ checks to see if the operand is negative. Is_null will check to see if the operand is null, or if there is a blank. It is also used as a flag for the opcodes target, if the opcode is just full of blank spaces. Value_ will take the actual 'int' translation of the hex value. Error_messages_ creates the error messages given to the user if any of the values are invalid. Text_ will be the actual source text from the file of the operand.

Main Class:

This class will take care of running the program and setting all the arguments of what needs to be inputted into the program. Arg 1 will take care of the file that is being inputted. Arg 2 will write to a binary file first, then it will write to a text file. Arg 3 will create a log file of what has been executed. The input and output streams are also setup in this file, in_scanner and out_stream.

OneMemoryWord Class:

This class will be used as a container for one word of memory. The memory word consists of a 16-bit value. The class will format the bits and print them as 3 + 1 + 12, corresponding to the opcode, whether or not it is indirect, and the address of the code. Different accessors are setup to

get different parts of the 16-bit pattern. Bit_pattern_ is used to get the full 16-bit number from the source file. Address_bits_ is used to get the last 12 bits that correspond to the memory address of the opcode. Indirect_flag_ looks at the fourth bit and determines if it is a direct or indirect reference, see codeline class. Mnemonic_bits_ looks at the first 3 numbers of the 16-bit number, and those numbers correspond to the actual opcode, ADD, LD, RD, WRT.

Pullet16Assembler Class:

This class will be used to assemble the code. In_scanner and out_stream will be used to read and write from the source files. This assembler is a two-pass assembler. The first pass will produce the symbol table and check for errors in the symbols provided in the source file. Errors would include coming across multiply defined symbols. Pass two will generate the machine code, that would be the instructions to the CPU. Both passes will pull the necessary values from the codeline class. The results would then be dumped out to the user. We have also forced symbols and mnemonics to have blank spaces at the end of them, so that they will have be the same length. Both symbols and mnemonics consist of three characters. Functions will also be used to print out the code lines of the code pattern, print out the machine code, and to print out the symbol table. Our program counter will also need to be updated after these functions, as we need to keep track of where we are in memory. There is also a function to update the symbol table after each pass. The default value of the table is 0. This means that we are unable to store a value at location 0. We will add a value to memory, then back out once the first pass of the assembler is complete. Error messages are also setup in this class that correspond to a bad symbol, hex operator, and a mistype of words or text.

Symbol Class:

This class acts as a container for keeping one symbol found in the source file. First the location of the symbol is stored into memory. The symbol will then be checked to see if it is valid. This is done by checking to see if the first character is an alpha symbol. Then we will check to see if the second and third characters are alphanumeric. This class will also check for errors while scanning through the source file being read. The function is_multiply will also check to see if the same symbol comes up more than once in the file. Only once instance of the symbol will be stored into memory.

ToStrings:

ToStrings will be setup throughout each class and will be used to print out any values that each class obtains. Any formatting that needs to be addressed will be done in these ToSting functions.