**Modified Sic One Pass Assembler**

1. **Team members:**

* Nada Hamada 20101043 Class 5
* Ahmed Elshennawy 20100483 Class 7
* Ahmed Amr 20100159 Class 5

1. **Detailed Description of the Code and Functions Used:**

This code is a one pass assembler that works on modified sic assembly language. Modified sic includes format 3, format 3 modified - which deals with immediate values, and format 1 instructions.

A one pass assembler generates the location counter and object code of an instruction while generating a corresponding text record and symbol table in one process.

Location counters are incremented according to the instruction format. Unless the instruction is a byte/word/resw/resb, these have special calculations which are covered in the code.

The object code is made up of 2 parts, opcode - result of mapping the instruction to the dictionary, and a target label - address/immediate value. In case of an immediate value, it is written as 4 hex characters. I case of an address; it could be referencing a line whose address is known (backward referencing) and therefore found in the symbol table so the target label is 4 hex characters of that address; or it could be referencing a line whose location is yet to be calculated (forward referencing). If an instruction is referencing an upcoming line, the target label is 4 0’s, and the address of the bytes to be modified are added to a linked list so that when the line is reached, all the bytes where that address was supposed to be is modified. If this label was a combination of an address and an indexing register, a flag is raised so that during modification the target label is updated to be address + 0x8000.

To achieve this, a series of steps is required.

4 dictionaries are defined, 3 of them are for the instructions which map each key (instruction) to its corresponding opcode. The 4th dictionary contains the hexadecimal ascii code of uppercase and lowercase letters, numbers, and special characters.

Input file’s (assembly) path is defined, and any output files are created during runtime.

**Class Node** is the linked list element definition. It is unlike the usual node definition as it contains “value” and “index”. Value will contain the address to be modified. Index is a flag, which is set when the address has an indexing register.

**Print\_linked\_list** is a function that loops through the linked list and prints all values in it.

**Remove\_index** is a function that extracts the target address from a label containing an indexing register if present. For example, BUFFER,X returns BUFFER.

**Remove\_immediate** is function that removes an immediate value from a label. BUFFER,#4 returns BUFFER. We are not entirely sure if sic assembly allows such kind of a target address, however we’ve handled it either way.

**Remove\_hash** is a function that removes the hashtag from an immediate value so that we can process the value as an integer. #4 returns 4.

**Check\_target\_label** is function that takes 4 parameters - symbol table, label, location counter, index. As we’ve mentioned before index is a flag that indicates whether the label originally had an indexing register. This function works by mapping the target label to the symbol table dictionary. The result could be one of two things, the target label has been previously added to the dictionary (could be defined by a backward reference in this case it has an integer value, or forward reference defined by a linked list), or this is the first time it is being referenced (in this case added to the symbol table dictionary as a linked list).

**Check\_line\_label** is function that maps a line label to the symbol table, it is either found to be a linked list - in this case a node is added to the beginning of the list containing the location counter of this line; or it is not found - in this case it is defined as integer.

Processing the assembly code itself works by reading the input file line by line. The first line helps us generate part of the header, as it gives us the programme name and starting address. However, the header record is written once we reach the end so that we could calculate the size of the programme.

The rest of the lines are read through a for loop. Each line is split into elements separated by white spaces. Depending on the number of elements in the line, it is processed accordingly. The number of elements could either be 3, 2, or 1.

**3 Elements:** the line has a line label, instruction, and target label. In such a case, we need to map the line label to the symbol table by using the chack\_line\_label function. The instruction is initially mapped to the format 3 dictionary since format 1 instructions don’t take target addresses. The target label is then processed using the functions we’ve created to check whether it is a simple label, contains an index register, or an immediate value. In the case of an immediate value the instruction is mapped to the modified dictionary. If the mapping to format 3 is unsuccessful, the instruction is tested to be word/byte/resw/resb.

**2 Elements:** the line is either format 1 and a line label, or format 3 (may be modified, too) and target label. The 2nd element (instruction) is first mapped to format 1 dictionary. If the mapping is unsuccessful it is then mapped to format 3 instructions and processed the same way as if they were 3 elements in the line (without the possibility of word/byte/resw/resb as these instructions require line and target labels).

**1 Element:** this is only an instruction, which could only be format 1, RSUB format 3 instruction, or END. Format 1 and 3 instructions have already been discussed. If the instruction was END, the header record is written, the size is calculated, the text records are copied from t.txt file to hte.txt file hen the end record is written which contains the address of the 1st executable instruction. To find this address, a counter is incremented every time an instruction is mapped, then verified if the counter is == 1, if true the location address of this instruction is stored in a variable.

1. **The Code:**

Please note that is not a screenshot, this is the copied code from PyCharm.

# creating the instruction set dictionary for format 3 without modification  
instruction\_set\_for3 = {  
 'ADD': 0x18, 'AND': 0x40, 'COMP': 0x28,  
 'DIV': 0x24, 'J': 0x3C, 'JEQ': 0x30,  
 'JGT': 0x34, 'JLT': 0x38, 'JSUB': 0x48,  
 'LDA': 0x00, 'LDCH': 0x50, 'LDL': 0x08,  
 'LDX': 0x04, 'MUL': 0x20, 'OR': 0x44,  
 'RD': 0xD8, 'RSUB': 0x4C, 'STA': 0x0C,  
 'STCH': 0x54, 'STL': 0x14, 'STSW': 0xE8,  
 'STX': 0x10, 'SUB': 0x1C, 'TD': 0xE0,  
 'TIX': 0x2C, 'WD': 0xDC,  
}  
  
# creating the instruction set dictionary for format 3 with modification (immediate operand)  
instruction\_set\_for3\_modi = {  
 'ADD': 0x19, 'AND': 0x41, 'COMP': 0x29,  
 'DIV': 0x25, 'J': 0x3D, 'JEQ': 0x39,  
 'JGT': 0x35, 'JLT': 0x39, 'JSUB': 0x49,  
 'LDA': 0x01, 'LDCH': 0x51, 'LDL': 0x09,  
 'LDX': 0x05, 'MUL': 0x21, 'OR': 0x45,  
 'RD': 0xD9, 'STA': 0x0D,  
 'STCH': 0x55, 'STL': 0x15, 'STSW': 0xE9,  
 'STX': 0x11, 'SUB': 0x1D, 'TD': 0xE1,  
 'TIX': 0x2D, 'WD': 0xDD,  
}  
  
# creating the instruction set dictionary for format 1  
instruction\_set\_for1 = {  
 'FIX': 0xC4, 'FLOAT': 0xC0,  
 'HIO': 0xF4, 'NORM': 0xC8,  
 'SIO': 0xF0, 'TIO': 0xF8,  
}  
  
# ascii code dictionary for characters  
ascii\_dictionary = {  
 # upper case letters  
 'A': 0x41, 'B': 0x42, 'C': 0x43, 'D': 0x44, 'E': 0x45,  
 'F': 0x46, 'G': 0x47, 'H': 0x48, 'I': 0x49, 'J': 0x4A,  
 'K': 0x4B, 'L': 0x4C, 'M': 0x4D, 'N': 0x4E, 'O': 0x4F,  
 'P': 0x50, 'Q': 0x51, 'R': 0x52, 'S': 0x53, 'T': 0x54,  
 'U': 0x55, 'V': 0x56, 'W': 0x57, 'X': 0x58, 'Y': 0x59,  
 'Z': 0x5A,  
  
 # lower case letters  
 'a': 0x61, 'b': 0x62, 'c': 0x63, 'd': 0x64, 'e': 0x65,  
 'f': 0x66, 'g': 0x67, 'h': 0x68, 'i': 0x69, 'j': 0x6A,  
 'k': 0x6B, 'l': 0x6C, 'm': 0x6D, 'n': 0x6E, 'o': 0x6F,  
 'p': 0x70, 'q': 0x71, 'r': 0x72, 's': 0x73, 't': 0x74,  
 'u': 0x75, 'v': 0x76, 'w': 0x77, 'x': 0x78, 'y': 0x79,  
 'z': 0x7A,  
  
 # numbers  
 '0': 0x30, '1': 0x31, '2': 0x32, '3': 0x33, '4': 0x34,  
 '5': 0x35, '6': 0x36, '7': 0x37, '8': 0x38, '9': 0x39,  
  
 # special characters  
 ' ': 0x20, '!': 0x21, '"': 0x22, '#': 0x23, '$': 0x24,  
 '%': 0x25, '&': 0x26, "'": 0x27, '(': 0x28, ')': 0x29,  
 '\*': 0x2A, '+': 0x2B, ',': 0x2C, '-': 0x2D, '.': 0x2E,  
 '/': 0x2F, ':': 0x3A, ';': 0x3B, '<': 0x3C, '=': 0x3D,  
 '>': 0x3E, '?': 0x3F, '@': 0x40, '[': 0x5B, '\\': 0x5C,  
 ']': 0x5D, '^': 0x5E, '\_': 0x5F, '`': 0x60, '{': 0x7B,  
 '|': 0x7C, '}': 0x7D, '~': 0x7E  
}  
  
# create the assembly text file, output will be with location counter and machine codes  
assembly\_code = 'assembly\_code.txt'  
assembly\_code\_file = open(assembly\_code, 'w')  
  
# read the assembly.txt file  
assembly = f'assembly.txt'  
  
# create a symbol\_table.txt file (shows steps)  
symbol\_table\_file = open('symtable\_steps.txt', 'w')  
  
# create a symtable.txt file (final)  
symbol\_table\_final = open('symbol\_table.txt', 'w')  
  
# create the t.txt file  
t\_record = open('t.txt', 'w')  
  
# create the hte\_record.txt file  
hte\_record = open('hte.txt', 'w')  
  
  
# defining a new data type, nodes of the linked list  
class Node:  
 def \_\_init\_\_(self, value, index):  
 self.value = value  
 self.next = None  
 self.index = index  
  
  
# function to print elements of a linked list  
def print\_linked\_list(symbol\_table, label):  
 # retrieve the first node in the linked list from the symbol table  
 node = symbol\_table[label]  
 # loop through the linked list and print the elements  
 while node:  
 symbol\_table\_file.write(f"{label} : {node.value:04X}\n")  
 node = node.next  
  
  
# function to remove an indexing register from a label so that it doesn't interfere with searching for the label in the dictionary  
def remove\_index(label):  
 # searches for index in a label  
 if ",X" or ",x" or "x," or "X," in label:  
 new\_label = label.replace(",X" or ",x" or "x," or "X,", "")  
 return new\_label  
 else:  
 return label  
  
  
# library which we will use to remove digits (immediate values) from a string  
import re  
  
  
# function to remove an immediate value from a label so that it doesn't interfere with searching for the label in the dictionary  
def remove\_immediate(label):  
 # Remove commas  
 inter = label.replace(",", "")  
 # remove #  
 new\_label = inter.replace("#", "")  
 # Remove digits  
 new\_label = re.sub(r'\d', '', new\_label)  
  
 # this condition check uf the string is now empty, which means it was only an immediate value, no target address involved  
 if new\_label == '':  
 # this string is returned to verify what the label originally was within the code  
 return "EMPTY"  
 else:  
 # the string was not just an immediate value  
 return new\_label  
  
  
# function that remoes the # from an immediate value, we will need it zin order to process the number alone  
def remove\_hash(label):  
 # remove #  
 new\_label = label.replace("#", "")  
 return new\_label  
  
  
# function that searches for a given target label/address in the symbol table, process it as required  
def check\_target\_label(symbol\_table, label, counter, index):  
 # check for occurrences in the symbol table dictionary  
 if label in symbol\_table:  
 value = symbol\_table[label]  
 # checks whether the given target label has been defined before trying to address it (represented by a number for the target location)  
 if isinstance(value, int):  
 print(f"The value of {label} is a number.")  
  
 # checks whether the given target label is yet to be defined before trying to address it, but has already been targeted before (represented by a linked list for the target location until it is found)  
 elif isinstance(value, Node):  
 print(f"The value of {label} is a linked list. Adding {(counter + 1):04X} to the list.") # +1 because we will want to modify the address only in the object code, skipping the first byte (opcode)  
 # create a node in the linked list and give it the current location counter + 1, since we will want to modify the address part of the object code only  
 new\_node = Node(counter + 1, index)  
 new\_node.next = value.next  
 # set the index flag  
 # add the node to the end of the list by setting the pointer of the previous node's next to the new node  
 value.next = new\_node  
  
 # update symbol table file and print all occurrences of this label so far  
 symbol\_table\_file.write(f"\nNew location added to {label}\n")  
 print\_linked\_list(symbol\_table, label)  
 symbol\_table\_file.write(f"\n")  
  
 # adds the target label to the symbol as a linked list, because it hasn't been accessed before and yet to be defined  
 # line.strip() incase the label was index,immediate, an empty string would be returned after removal of index register and immediate values  
 elif label.strip():  
 print(f"{label} not found in the symbol table. Adding with value {counter + 1:04X}.")  
 # adds the label to the symbol table dictionary, create the first node in the linked list and give it the current location counter  
 symbol\_table[label] = Node(counter + 1, index)  
  
 # update symbol table file with the new addition  
 symbol\_table\_file.write(f"\nNew location added to {label}\n")  
 print\_linked\_list(symbol\_table, label)  
 symbol\_table\_file.write(f"\n")  
  
  
# function that searches for the line label in the symbol table, so that if it is a linked list, the address is updated  
def search\_line\_label(symbol\_table, label, counter):  
 # searches for this line label in the symbol table dictionary  
 if label in symbol\_table:  
 value = symbol\_table[label]  
  
 # checks if the line label was previously targeted but yet to be defined, value is of type linked list (node)  
 if isinstance(value, Node):  
 print(  
 f"The value of {label} is a linked list. Adding a new element at the beginning with the value {counter:04X}.")  
 # Create a new node with the current counter value  
 new\_node = Node(counter, 0)  
 # Set the next pointer of the new node to the current first node in the linked list  
 new\_node.next = value  
 # Set the linked list to the new node  
 symbol\_table[label] = new\_node  
 symbol\_table\_file.write(f"\nLine location of {label} found\n")  
 print\_linked\_list(symbol\_table, label)  
 symbol\_table\_file.write(f"\n")  
 symbol\_table\_final.write(f"{label}: {counter:04X}\n")  
  
 # if it isn't found then it is a new label yet to be accessed, define it  
 else:  
 print(f"{label} not found in the symbol table. Adding with value {counter:04X}.")  
 symbol\_table[label] = counter  
 line\_labels[label] = counter  
 # writes in both symbol table files  
 symbol\_table\_file.write(f"{label} : {symbol\_table[label]:04X}\n")  
 symbol\_table\_final.write(f"{label}: {symbol\_table[label]:04X}\n")  
  
  
# defining the symbol table dictionary  
symbol\_table = {}  
# specifically for line labels to easily write them in the symbol table file without having to check for the type of value (node or int) in the symbol table dictionary  
line\_labels = {}  
  
try:  
 # assembly -> input file, code to be read and assembled  
 # assembly\_code -> assembly\_code\_file, file to created, with location counters and machine codes  
 # symbol\_table -> symtable, file to be written in all the updates of the symbol table dictionary as we go  
 # symtable -> symbol\_table\_final, final symbol table  
 # t -> t\_record, file that will contain all t records until the rest of the code is processed  
 # hte -> hte\_record, output  
 with open(assembly, 'r') as input\_file, open(assembly\_code, 'w') as assembly\_code\_file, open('symtable\_steps.txt','w') as symtable, open('symbol\_table.txt', 'w') as symbol\_table\_final, open('t.txt', 'w') as t\_record, open('hte.txt','w') as hte\_record:  
 # Read the first line  
 first\_line = input\_file.readline().strip()  
  
 # Split the first line into elements  
 elements = first\_line.split()  
  
 prog\_name = elements[0]  
  
 # read the starting location of the code, 3rd element of the first line as per sic format: PROGNAME START LOC  
 start\_loc = elements[2]  
 loc = int(start\_loc, 16)  
 # size of the code yet to be calculated  
 size\_of\_prog = 0  
  
 first\_instruc\_loc = " "  
  
 assembly\_code\_file.write(f"LOC\tLABEL\t\tINSTRUC\t\tTARGET\t\tOBJECT\_CODE\tRELOC\n\n")  
 # write the first line from the input fie as it is into the complete assembly code file  
 assembly\_code\_file.write(f"\t{first\_line}\n")  
  
 # machine codes of current text record  
 machine\_code\_string = ""  
 # machine code of current instruction  
 machine\_code = ""  
  
 # reloc bit string for each text record  
 reloc\_bits = ""  
  
 # reloc bit for every instruction  
 reloc = 0  
  
 # counter to mark the first instruction (since we don't know which format the first instruction will be, the counter will be incremented at each type, then check if it is equal to 1, if so, the location address is stored to be written in the hte file)  
 instruc\_count = 0  
  
 # counter for the current t\_record size  
 t\_size = 0  
  
 # flag to indicate whether there is a byte/format 1 in the text record so that we would start w new text record once an instruction needs relocation  
 flag = 0  
  
 # read the rest of the file  
 file = input\_file.readlines()  
 # process through it line by line  
 for lines in file:  
  
 # read the current line  
 line = lines.strip()  
 print("\n", "-------------", line, "-------------")  
 # split it into elements (based on the empty spaces before and after a character)  
 elements = line.split()  
  
 # Algorithm to check the number elements in the line  
  
 # 3 elements means LINE\_LABEL INSTRUCTION TARGET\_LABEL  
 if len(elements) == 3:  
  
 # read the 1st element: line label  
 line\_label = elements[0]  
 # define the line and target labels in symbol table  
 search\_line\_label(symbol\_table, line\_label, loc)  
  
 # read the 2nd element: instruction  
 instruc = elements[1]  
 # map the instruction to format 3 instruction set dictionary  
 opcode = instruction\_set\_for3.get(elements[1])  
  
 # read the 3rd element: target label  
 target\_label = elements[2]  
  
 # check if this line label was previously forward referenced  
 if isinstance(symbol\_table[line\_label], Node):  
 # get the first node (contains modification amount)  
 modification\_value = symbol\_table[line\_label]  
 # get the next node, first address to be modified  
 node = symbol\_table[line\_label].next  
  
 # checks if there were any relocation bits  
 if reloc\_bits:  
 # fits them into 12 bits  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = int(reloc\_bits, 2)  
  
 # checks if there were any instructions to print them before writing the modification record  
 if t\_size > 0:  
 # write the current text record  
 t\_record.write(f"{t\_size:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
  
 # loops as long as there are more addresses to be modified  
 while node:  
 # checks the index flag for indexing register present inorder to modify the value accordingly  
 if node.index == 0:  
 t\_record.write(f"T.{node.value:06X}.02.000.{(modification\_value.value):04X}.\n")  
 else:  
 t\_record.write(f"T.{node.value:06X}.02.000.{(modification\_value.value + 32768):04X}.\n")  
 node = node.next  
  
 # this section will check if the next line has 3 elements, which may modify possible forward references  
 # Get the current position of the cursor  
 current\_position = input\_file.tell()  
 # Read the next line without changing the cursor position  
 next\_line = input\_file.readline()  
 # Count the number of elements in the next line  
 elements\_count = len(next\_line.split())  
 # Move the cursor back to the original position  
 input\_file.seek(current\_position)  
  
 # start new text record with current location counter  
 if elements\_count != 3:  
 # start the new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # reset text record size  
 t\_size = 0  
 # reset relocation string  
 reloc\_bits = ""  
 # reset machine code string  
 machine\_code\_string = ""  
 # reset byte flag  
 flag = 0  
  
 # check the result of the mapping  
 if opcode is not None:  
  
 # condition that searches for he loc of the first instruction  
 instruc\_count += 1  
 if instruc\_count == 1:  
 first\_instruc\_loc = loc  
  
 # no indexing, no immediate  
 if (remove\_index(target\_label)) == target\_label and (remove\_immediate(target\_label)) == target\_label:  
  
 # format 3 with target addresses require relocation  
 reloc = 1  
  
 # instruction is of format 3  
 print(f"\nMapped to instruction\_set\_for3\n{instruc}:{hex(opcode)}\n")  
  
 # define the line and target labels in symbol table  
 check\_target\_label(symbol\_table, remove\_index(target\_label), loc, 0)  
  
 # forward referencing  
 if isinstance(symbol\_table[target\_label], Node):  
 print(f"Machine Code: {opcode:02X}0000")  
 # write into complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\t\t{opcode:02X}0000\t\t{reloc}\n")  
 machine\_code = f"{opcode:02X}0000."  
  
 # backward referencing  
 else:  
 print(f"Machine Code: {opcode:02X}{symbol\_table[target\_label]:04x}")  
 # write into complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\t\t{opcode:02X}{symbol\_table[target\_label]:04X}\t\t{reloc}\n")  
 machine\_code = f"{opcode:02X}{symbol\_table[target\_label]:04X}."  
  
 # immediate only  
 elif remove\_immediate(target\_label) == "EMPTY":  
  
 # format 3 modified does not require relocation  
 reloc = 0  
  
 # map to the correct opcode now that we now it is format 3 modified after checking the label  
 opcode = instruction\_set\_for3\_modi.get(instruc)  
  
 # instruction is of format 3 modified  
 print(f"Machine Code: {opcode:02X}{int(remove\_hash(target\_label), 10):04X}")  
  
 # write into complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t\t{line}\t\t{opcode:02X}{int(remove\_hash(target\_label), 10):04X}\t\t{reloc}\n")  
 machine\_code = f"{opcode:02X}{int(remove\_hash(target\_label), 10):04X}."  
  
 # indexing + label  
 else:  
  
 # format 3 with target addresses require relocation  
 reloc = 1  
  
 # define the target label in symbol table  
 check\_target\_label(symbol\_table, remove\_index(target\_label), loc, 1)  
  
 # forward referencing  
 if isinstance(symbol\_table[remove\_index(target\_label)], Node):  
 print(f"Machine Code: {opcode:02X}0000")  
 # write into complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t{opcode:02X}0000\t\t{reloc}\n")  
 machine\_code = f"{opcode:02X}0000."  
  
 # backward referencing  
 else:  
 print(f"Machine Code: {opcode:02X}{(symbol\_table[remove\_index(target\_label)] + 0X8000):04x}")  
 # write into complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t{opcode:02X}{(symbol\_table[remove\_index(target\_label)] + 0X8000):04X}\t\t{reloc}\n")  
 machine\_code = f"{opcode:02X}{(symbol\_table[remove\_index(target\_label)] + 0X8000):04X}."  
  
 # increment location counter by 3 bytes  
 loc += 3  
 # increment t record size by 3 bytes  
 t\_size += 3  
  
 # check if this instruction fits in this t record, if not start a new one  
 if (t\_size > 30):  
  
 # append 2 zeros since we only have 10 out of 12 bits for the 3 hex values  
 reloc\_bits += "00"  
 # convert into integer equivalent  
 reloc\_bits = int(reloc\_bits, 2)  
  
 # write the current text record  
 t\_record.write(f"{(t\_size - 3):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start the new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # add the size of this instruction to the new text record  
 t\_size = 3  
 # add the machine code of this instruction to the new text record  
 machine\_code\_string = machine\_code  
 # reset the relocation string to the relocation bit of this instruction to the new text record  
 reloc\_bits = str(reloc)  
  
 # check if there is a byte in the t record  
 elif flag == 1:  
  
 # fit relocation string to 12 binary digits by appending as many zeros as necessary  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 # write the current text record without current instruction  
 t\_record.write(f"{t\_size-3:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start a new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # set machine code for this variable  
 machine\_code\_string = machine\_code  
 # append reloc bit to the string  
 reloc\_bits = str(reloc)  
 # add instruction size to text record  
 t\_size = 3  
 # reset byte flag  
 flag = 0  
  
 else:  
 # append the relocation bit to the current relocation string of this text record  
 reloc\_bits += str(reloc)  
 # append the current machine code to the string of machine codes in this text record  
 machine\_code\_string += machine\_code  
  
 # WORD is tested for outside of instruction set because it does not have an opcode  
 elif instruc == 'WORD':  
  
 flag = 0  
  
 # WORD instructions do not need relocation as they do not deal with addressing  
 reloc = 0  
  
 # adds the line label to the symbol table  
 search\_line\_label(symbol\_table, line\_label, loc)  
  
 print(f"Machine Code: {int(target\_label, 10):06X}")  
 # write in complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\t\t{int(target\_label, 10):06X}\t\t{reloc}\n")  
 machine\_code = f"{int(target\_label, 10):06X}."  
  
 # increment location counter by 3 bytes  
 loc += 3  
 # increment size of current text record by 3 bytes  
 t\_size += 3  
  
 # check if this instruction is more than the allowed capacity for one text record  
 if (t\_size > 30):  
  
 # append 2 zeros since we only have 10 out of 12 bits for the 3 hex values  
 reloc\_bits += "00"  
 # convert into integer equivalent  
 reloc\_bits = int(reloc\_bits, 2)  
  
 # write the current text record  
 t\_record.write(f"{(t\_size - 3):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start the new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # add the size of this instruction to the new text record  
 t\_size = 3  
 # add the machine code of this instruction to the new text record  
 machine\_code\_string = machine\_code  
 # reset the relocation string to the relocation bit of this instruction to the new text record  
 reloc\_bits = "0"  
  
 # check if there is a byte in the t record  
 elif flag == 1:  
  
 # fit relocation string to 12 binary digits by appending as many zeros as necessary  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 # write the current text record without this instruction  
 t\_record.write(f"{t\_size-3:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start a new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # set machine code for this variable  
 machine\_code\_string = machine\_code  
 # append reloc bit to the string  
 reloc\_bits = str(reloc)  
 # add the size of this instruction to the text record  
 t\_size = 3  
 # reset byte flag  
 flag = 0  
  
 else:  
 # append the relocation bit to the current relocation string of this text record  
 reloc\_bits += str(reloc)  
 # append the current machine code to the string of machine codes in this text record  
 machine\_code\_string += machine\_code  
  
 # RESW is tested for outside of instruction set because it does not have an opcode  
 elif instruc == 'RESW':  
 # adds the line label to the symbol table  
 search\_line\_label(symbol\_table, line\_label, loc)  
  
 # write in complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\n")  
  
 # increment location counter by reserve amount after conversion to bytes  
 loc += int(elements[2]) \* 3  
  
 # fit relocation string to 12 binary digits by appending as many zeros as necessary  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 # write the current text record  
 t\_record.write(f"{t\_size:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start the new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # reset text record size  
 t\_size = 0  
 # reset relocation string  
 reloc\_bits = ""  
 # reset machine code string  
 machine\_code\_string = ""  
 # reset byte flag  
 flag = 0  
  
 # RESB is tested for outside of instruction set because it does not have an opcode  
 elif instruc == 'RESB':  
 # adds the line label to the symbol table  
 search\_line\_label(symbol\_table, line\_label, loc)  
  
 # write in complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\n")  
  
 # increment location counter by reserve amount in bytes  
 loc += int(elements[2])  
  
 # fit relocation string to 12 binary digits by appending as many zeros as necessary  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 # write the current text record  
 t\_record.write(f"{t\_size:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start the new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # reset text record size  
 t\_size = 0  
 # reset relocation string  
 reloc\_bits = ""  
 # reset machine code string  
 machine\_code\_string = ""  
 # reset th byte flag  
 flag = 0  
  
 # BYTE is tested for outside of instruction set because it does not have an opcode  
 elif instruc == 'BYTE':  
  
 flag = 1  
  
 # BYTE instructions do not need relocation as they do not deal with addressing  
 # due to their size, it causes complications with the following instructions and their relocation, therefore it is the last object code in a text record  
 reloc = 0  
  
 # adds the line label to the symbol table  
 search\_line\_label(symbol\_table, line\_label, loc)  
  
 # check if the value is a string  
 if elements[2][0] == 'C':  
 ascii\_code = ""  
 for char in elements[2][2:-1]:  
 ascii\_code += f'{ascii\_dictionary.get(char):X}'  
  
 print(f"Machine Code: {ascii\_code}")  
  
 # write in complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\t\t{ascii\_code}\t\t{reloc}\n")  
 machine\_code = f"{ascii\_code}."  
  
 # minus 3 because of C''  
 loc += len(elements[2]) - 3  
 # increase text record with corresponding size  
 t\_size += len(elements[2]) - 3  
  
 # check if the value is a hexadecimal value  
 elif elements[2][0] == 'X':  
 # write in complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\t\t{elements[2][2:-1]}\t\t{reloc}\n")  
 machine\_code = f"{elements[2][2:-1]}."  
  
 # divided by 2 because 2 hex characters represent 1 byte  
 loc += (len(elements[2]) - 3) // 2  
 # increase the text record with corresponding size  
 t\_size += (len(elements[2]) - 3) // 2  
  
 # append relocation bit to string  
 reloc\_bits += str(reloc)  
  
 if (t\_size > 30):  
  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 print(reloc\_bits)  
  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 t\_record.write(f"{(t\_size - 1):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 t\_record.write(f"T.{loc:06X}.")  
  
 t\_size = 1  
 machine\_code\_string = machine\_code  
 reloc\_bits = str(reloc)  
 flag = 0  
  
 else:  
 reloc\_bits += str(reloc)  
 machine\_code\_string += machine\_code  
  
 # 2 elements means INSTRUCTION(format 3) TARGET\_LABEL or LINE\_LABEL INSTRUCTION(format 1)  
 elif len(elements) == 2:  
 # read the 1st element: instruction  
 instruc = elements[0]  
  
 # map it to format 1  
 opcode = instruction\_set\_for1.get(elements[0])  
  
 # read the 2nd element: target label  
 target\_label = elements[1]  
  
 # check the result of the mapping  
 if opcode is not None:  
 # condition that searches for he loc of the first instruction  
 instruc\_count += 1  
 if instruc\_count == 1:  
 first\_instruc\_loc = loc  
  
 # fromat 1 instruction are for the assembler and o not deal with the memory, therefore they do not require relocation  
 # due to their size they are the last object code to be written in a text record  
 reloc = 0  
  
 flag = 1  
  
 print(f"\nMapped to instruction\_set\_for1\n{instruc}:{hex(opcode)}\n")  
 search\_line\_label(symbol\_table, line\_label, loc)  
 assembly\_code\_file.write(f"{loc:04X}\t{line}\t\t\t\t\t\t{opcode:02X}\t\t{reloc}\n")  
 machine\_code = f"{opcode:02X}."  
  
 loc += 1  
 t\_size += 1  
  
 if (t\_size > 30):  
  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 print(reloc\_bits)  
  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 t\_record.write(f"{(t\_size - 1):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 t\_record.write(f"T.{loc:06X}.")  
  
 t\_size = 1  
 machine\_code\_string = machine\_code  
 reloc\_bits = str(reloc)  
  
 else:  
 reloc\_bits += str(reloc)  
 machine\_code\_string += machine\_code  
  
 # map element to instruction set format 3  
 else:  
  
 opcode = instruction\_set\_for3.get(elements[0])  
  
 if opcode is not None:  
  
 flag = 0  
  
 instruc\_count += 1  
 if instruc\_count == 1:  
 first\_instruc\_loc = loc  
  
 # no indexing, no immediate  
 if (remove\_index(target\_label)) == target\_label and (remove\_immediate(target\_label)) == target\_label:  
  
 reloc = 1  
  
 # instruction is of format 3  
 print(f"\nMapped to instruction\_set\_for3\n{instruc}:{hex(opcode)}\n")  
  
 # define the line and target labels in symbol table  
 check\_target\_label(symbol\_table, remove\_index(target\_label), loc, 0)  
  
 # forward referencing  
 if isinstance(symbol\_table[target\_label], Node):  
 print(f"Machine Code: {opcode:02X}0000")  
 machine\_code = f"{opcode:02X}0000."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t{opcode:02X}0000\t\t{reloc}\n")  
  
 # backward referencing  
 else:  
 print(f"Machine Code: {opcode:02X}{(symbol\_table[target\_label]):04X}")  
 machine\_code = f"{opcode:02X}{(symbol\_table[target\_label]):04X}."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t{opcode:02X}{symbol\_table[target\_label]:04X}\t\t{reloc}\n")  
  
 # immediate only  
 elif remove\_immediate(target\_label) == "EMPTY":  
  
 reloc = 0  
  
 opcode = instruction\_set\_for3\_modi.get(instruc)  
 print(f"Machine Code: {opcode:02X}{int(remove\_hash(target\_label), 10):04X}")  
 machine\_code = f"{opcode:02X}{int(remove\_hash(target\_label), 10):04X}."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t{opcode:02X}{int(remove\_hash(target\_label), 10):04X}\t\t{reloc}\n")  
  
 # indexing + label  
 else:  
 reloc = 1  
  
 check\_target\_label(symbol\_table, remove\_index(target\_label), loc, 1)  
 # forward referencing  
 if isinstance(symbol\_table[remove\_index(target\_label)], Node):  
 print(f"Machine Code: {opcode:02X}0000")  
 machine\_code = f"{opcode:02X}0000."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t{opcode:02X}0000\t\t{reloc}\n")  
  
 # backward referencing  
 else:  
 print(f"Machine Code: {opcode:02X}{(symbol\_table[remove\_index(target\_label)] + 0X8000):04x}")  
 machine\_code = f"{opcode:02X}{(symbol\_table[remove\_index(target\_label)] + 0X8000):04x}."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t{opcode:02X}{(symbol\_table[remove\_index(target\_label)] + 0X8000):04X}\t\t{reloc}\n")  
  
 # increment location counter by 3 bytes  
 loc += 3  
 t\_size += 3  
  
 if (t\_size > 30):  
  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 print(reloc\_bits)  
  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 t\_record.write(f"{(t\_size - 3):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 t\_record.write(f"T.{loc:06X}.")  
  
 t\_size = 3  
 machine\_code\_string = machine\_code  
 reloc\_bits = str(reloc)  
  
 elif flag == 1:  
 # fit relocation string to 12 binary digits by appending as many zeros as necessary  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 # write the current text record  
 t\_record.write(f"{t\_size - 3:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start a new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # set machine code for this variable  
 machine\_code\_string = machine\_code  
 # append reloc bit to the string  
 reloc\_bits = str(reloc)  
 t\_size = 3  
 flag = 0  
  
 else:  
 reloc\_bits += str(reloc)  
 machine\_code\_string += machine\_code  
  
 # 1 element means format 1 instruction with no line label, or RSUB with no line label  
 elif len(elements) == 1:  
 instruc = elements[0]  
  
 # END is printed as it is, both elements are labels, target label is of the first executable instruction  
 if instruc == 'END':  
 # write into complete assembly file  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\n")  
  
 # append reloc bit to the string  
 reloc\_bits += str(reloc)  
 # append zeros as much as needed to fit 12 bits  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
  
 reloc\_bits = int(reloc\_bits, 2)  
  
 # write the current text record  
 t\_record.write(f"{(t\_size):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
  
 print(f"\nStarting Location: {int(start\_loc, 16):04X}")  
 print(f"End Location: {loc:04X}")  
 # calculate size of the program  
 size\_of\_prog = loc - int(start\_loc, 16)  
 print(f"Length of Program: {size\_of\_prog:04X}")  
  
 # write the complete header record  
 hte\_record.write(f"H.{prog\_name: <6}.{int(start\_loc, 16):06X}.{size\_of\_prog:06X}\n")  
 hte\_record.write(f"T.{int(start\_loc, 16):06X}.")  
  
 t\_record.close()  
  
 with open("t.txt", "r") as t\_record:  
 # parse through all the t records written and write them into the final and complete hte file  
 t\_lines = t\_record.readlines()  
 for tline in t\_lines:  
 line = tline.strip()  
 if line[9:11] == "00":  
 continue  
 else:  
 hte\_record.write(line)  
 hte\_record.write("\n")  
  
 # writing the end record  
 hte\_record.write(f"E.{first\_instruc\_loc:06X}")  
  
 else:  
 # map to instruction set format 1  
 opcode = instruction\_set\_for1.get(elements[0])  
 if opcode is not None:  
  
 reloc = 0  
  
 instruc\_count += 1  
 if instruc\_count == 1:  
 first\_instruc\_loc = loc  
  
 print(f"\nMapped to instruction\_set\_for1\n{instruc}:{hex(opcode)}\n")  
 machine\_code = f"{opcode:02X}."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t\t\t{opcode:02X}\t\t{reloc}\n")  
  
 loc += 1  
 t\_size += 1  
 reloc\_bits += str(reloc)  
 machine\_code\_string += machine\_code  
 flag = 1  
  
 # RSUB with no line label  
 elif instruc == 'RSUB':  
  
 reloc = 0  
  
 instruc\_count += 1  
 if instruc\_count == 1:  
 first\_instruc\_loc = loc  
  
 opcode = instruction\_set\_for3.get(elements[0])  
 print("\nRSUB instruction with no line label\n")  
 machine\_code = f"{opcode:02X}0000."  
 assembly\_code\_file.write(f"{loc:04X}\t\t\t{line}\t\t\t\t{opcode:02X}0000\t\t{reloc}\n")  
 loc += 3  
 t\_size += 3  
  
 if (t\_size > 30):  
 reloc\_bits += "00"  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 t\_record.write(f"{(t\_size - 3):02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 t\_record.write(f"T.{loc:06X}.")  
  
 t\_size = 3  
 machine\_code\_string = machine\_code  
 reloc\_bits = str(reloc)  
  
 elif flag == 1:  
 # fit relocation string to 12 binary digits by appending as many zeros as necessary  
 if len(reloc\_bits) < 12:  
 reloc\_bits = reloc\_bits.ljust(12, '0')  
 # convert to integer equivalent  
 reloc\_bits = (int(reloc\_bits, 2))  
  
 # write the current text record  
 t\_record.write(f"{t\_size - 3:02X}.{reloc\_bits:03X}.{machine\_code\_string}\n")  
 # start a new text record  
 t\_record.write(f"T.{loc:06X}.")  
  
 # set machine code for this variable  
 machine\_code\_string = machine\_code  
 # append reloc bit to the string  
 reloc\_bits = str(reloc)  
 t\_size = 3  
 flag = 0  
  
 else:  
 reloc\_bits += str(reloc)  
 machine\_code\_string += machine\_code  
  
 input\_file.close()  
 assembly\_code\_file.close()  
 symtable.close()  
 hte\_record.close()  
 t\_record.close()  
  
 print(f"\nLine\_Labels dictionary:")  
 for key, value in line\_labels.items():  
 print(f"{key}: {value:04X}")  
  
 print(f"\nSymbol\_Table dictionary:")  
 for key, value in symbol\_table.items():  
 print(f"{key}: {value}")  
  
except FileNotFoundError:  
 print(f"Error: The file '{assembly}' was not found.")  
except Exception as e:  
 print(f"An error occurred: {e}")

1. **Output Screenshot:**

A screenshot of a computer

Description automatically generated