Modified Sic One Pass Assembler

1. Team members:

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

2. 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.

3. 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\tRE
LOC\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[tar
get_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_has
h(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{relo
c}\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_h
ash(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{relo
c}\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)
```

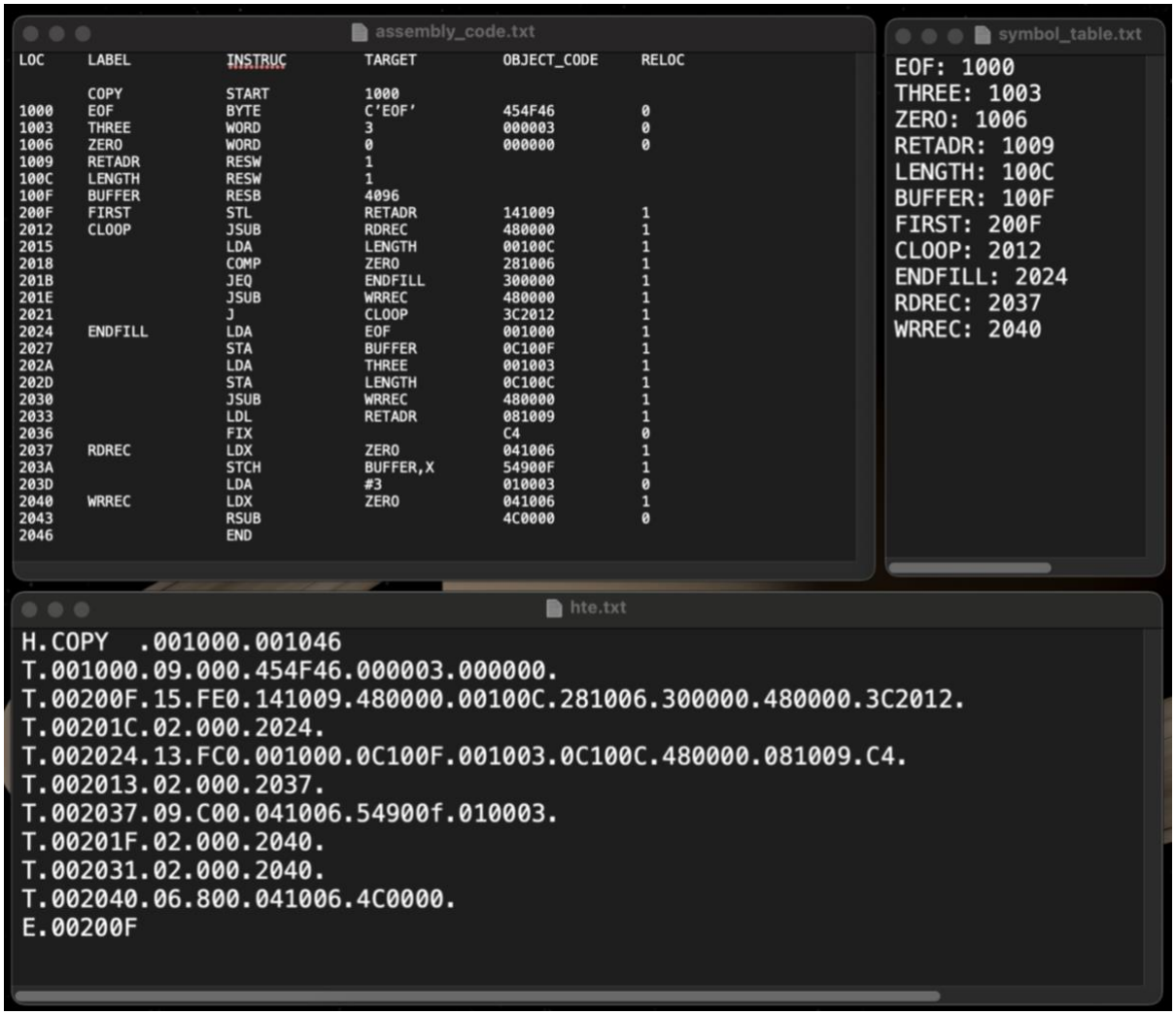
```
                        elif flag == 1:
                            # fit relocation string to 12 binary digits by
appending as many zeros as necessary
                            if len(reloc_bits) < 12:
                                reloc_bits = reloc_bits.ljust(12, '0')
                            # 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{relo
c}\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}")
```

4. Output Screenshot:



assembly_code.txt

LOC	LABEL	INSTRUC	TARGET	OBJECT_CODE	RELOC
	COPY	START	1000		
1000	EOF	BYTE	C'EOF'	454F46	0
1003	THREE	WORD	3	000003	0
1006	ZERO	WORD	0	000000	0
1009	RETADR	RESW	1		
100C	LENGTH	RESW	1		
100F	BUFFER	RESB	4096		
200F	FIRST	STL	RETADR	141009	1
2012	CLOOP	JSUB	RDREC	480000	1
2015		LDA	LENGTH	00100C	1
2018		COMP	ZERO	281006	1
201B		JEQ	ENDFILL	300000	1
201E		JSUB	WRREC	480000	1
2021		J	CLOOP	3C2012	1
2024	ENDFILL	LDA	EOF	001000	1
2027		STA	BUFFER	0C100F	1
202A		LDA	THREE	001003	1
202D		STA	LENGTH	0C100C	1
2030		JSUB	WRREC	480000	1
2033		LDL	RETADR	081009	1
2036		FIX		C4	0
2037	RDREC	LDX	ZERO	041006	1
203A		STCH	BUFFER,X	54900F	1
203D		LDA	#3	010003	0
2040	WRREC	LDX	ZERO	041006	1
2043		RSUB		4C0000	0
2046		END			

symbol_table.txt

```
EOF: 1000
THREE: 1003
ZERO: 1006
RETADR: 1009
LENGTH: 100C
BUFFER: 100F
FIRST: 200F
CLOOP: 2012
ENDFILL: 2024
RDREC: 2037
WRREC: 2040
```

hte.txt

```
H.COPY  .001000.001046
T.001000.09.000.454F46.000003.000000.
T.00200F.15.FE0.141009.480000.00100C.281006.300000.480000.3C2012.
T.00201C.02.000.2024.
T.002024.13.FC0.001000.0C100F.001003.0C100C.480000.081009.C4.
T.002013.02.000.2037.
T.002037.09.C00.041006.54900f.010003.
T.00201F.02.000.2040.
T.002031.02.000.2040.
T.002040.06.800.041006.4C0000.
E.00200F
```