Table of Contents

[ABSTRACT 6](#_Toc132296279)

[BUILDING OF THE ASSEMBLER 7](#_Toc132296280)

[Introduction 7](#_Toc132296281)

[CODING THE ASSEMBLER USING PYTHON 9](#_Toc132296282)

[CODE EXPLANATION 13](#_Toc132296283)

[RESULT 19](#_Toc132296284)

LIST OF TABLES:

[Table 1.Computation Table 10](#_Toc132374548)

[Table 2.Destination Table 11](#_Toc132374549)

[Table 3.Jump Table 11](file:///D:\SEM_2\EOC2\Assignment%204\ASSEMBLER_Report.docx#_Toc132374550)

LIST OF FIGURES:

[Figure 1.Symbolic Syntax: A-Instruction 8](#_Toc132374640)

[Figure 2.Symbolic syntax: C-Instruction 8](#_Toc132374641)

[Figure 3.Binary Syntax: C-Instruction 9](#_Toc132374642)

# ABSTRACT

Many stages are involved in the process of building a computer in order to convert a human thought into binary language, a language that the computer understands. Since we as coders code into the computer in machine language, which in our case is Hack, it is necessary to convert our Hack commands into binary form so that our computer can understand the commands and execute them. This process of conversion from Hack language to binary is done through an assembler programme that reads the Hack code and generates the corresponding binary code. The resulting binary code is then loaded into the computer's memory for execution.

The assembler is a type of programming language that is used to write instructions that can be executed by a computer's processor. Unlike high-level programming languages like Python or Java, assembler is a low-level language that is specific to the computer architecture and is closely related to the machine language.

Machine language is a set of binary instructions that a computer can understand and execute. Each instruction in machine language is represented by a series of binary digits, or bits, that the computer's processor can interpret as specific commands. These commands can be used to perform operations like adding numbers, moving data between memory locations, and branching to different parts of a program. Assembler is used to write programs in a more human-readable format than machine language, where, the programmer writes instructions using a series of mnemonic codes that represent specific machine language instructions. The assembler program then translates the mnemonic codes into the equivalent machine language instructions that the computer can execute. This translation process is known as assembly and is performed by a program called an assembler. Assemblers typically provide additional features such as the ability to define symbolic constants, labels, and macros to make it easier to write and maintain complex programs.

# 

# BUILDING OF THE ASSEMBLER

## Introduction

The machine language has two flavors: symbolic and binary. The binary code represents the actual machine language, as understood by the hardware. In machine language, symbols are essentially labels or names that are used to represent various elements of a program, such as instructions, data, or memory addresses. A programmer may also use variables to represent a specific value and the translator will automatically assign them to memory addresses. The programmer can also mark various locations in the program with symbols. One can declare labels to refer to a certain segment of the code.

To execute an assembly program, it must first be translated into binary language. This is done by the assembler. The assembler takes input as assembly language commands and converts it into binary codes. The resulting code can be loaded as is into the computer’s memory and executed by the hardware.

The programs in binary machine code and in assembly code are stored in text files with “hack” and “asm” extensions, respectively. Thus, a Prog.asm file is translated by the assembler into a Prog.hack file.

The binary code file is composed of text lines. Each line is a sequence of 16 “0”and “1” ASCII characters, coding a single 16-bit machine language instruction.

Assembly Language (.asm) Files An assembly language file is composed of text lines, each representing either an instruction or a symbol declaration. The instructions in the .asm or .hack files can be either an A-Instruction or a C-Instruction.

The constants must be non-negative and are written in decimal notation. A user-defined symbol can be any sequence of letters, digits, underscore (\_), dot (.), dollar sign ($), and colon (:)that does not begin with a digit. Comments are text beginning with two slashes (//) and ending at the end of the line is considered a commented is ignored. Space characters are ignored. Empty lines are ignored. All the assembly mnemonics must be written in uppercase. The rest which includes the user-defined labels and variable names are case sensitive. The convention is to use uppercase for labels and lowercase for variable names.

*A-Instructions*:

Syntax:

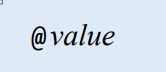


Figure 1.Symbolic Syntax: A-Instruction

Where the value is either a non-negative decimal constant or a symbol referring to such a constant.

In binary, the value of the @ is ‘0’, i.e., the most symbolic bit is 0 followed by the 15-bit value.

*C-Instructions*:

Symbolic Syntax:



Figure 2.Symbolic syntax: C-Instruction

The C-instruction must always have the computational bit, regardless of whether there is a destination or jump bit.

Binary Syntax:



Figure 3.Binary Syntax: C-Instruction

The first 3-bits in a C-Instruction is always 1. Depending on whether it is an A-register or a M-register, the ‘a’ bit is 0 or 1 respectively. The next 6-bits are the computational bits which are the outputs from the ALU. The next 3-bits are the destination bits, which specifies where to store the value. The last 3-bits are the jump instruction which specifies which instruction should be executed next.

Computational Table:

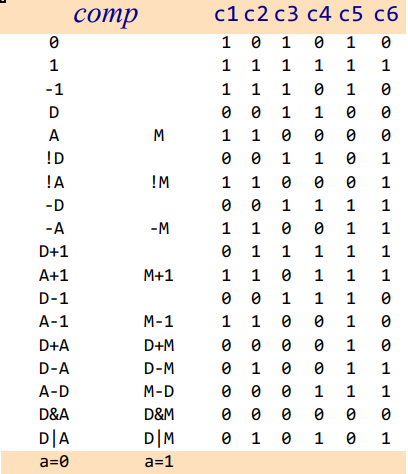


Table 1.Computation Table

In the code, we have considered all the possible permutations for the computational bits and have added the 0 or 1 for the ‘a’ bit with respect to whether the instruction is a A-instruction or M-instruction in the predefined table along with the computational bits.

Destination Table:

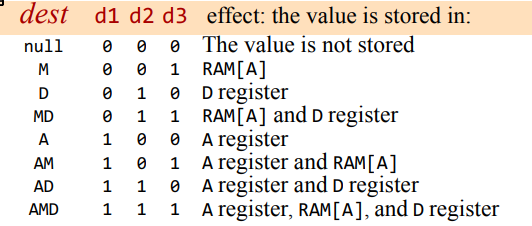


Table 2.Destination Table

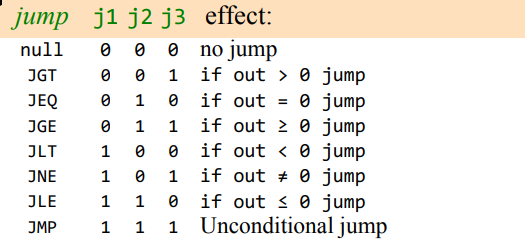


Table 3.Jump Table

Jump Table:

In the code, we have considered all the possible permutations for the destination and the jump bits.

# 

# CODING THE ASSEMBLER USING PYTHON

#Defining the values of computation instructions

computation\_dictionary = {

    "0" : "0101010",

    "1" : "0111111",

    "-1" : "0111010",

    "D" : "0001100",

    "A" : "0110000",

    "M" : "1110000",

    "!D" : "0001101",

    "!A" : "0110001",

    "!M" : "1110001",

    "-D" : "0001111",

    "-A" : "0110011",

    "-M" : "1110011",

    "D+1" : "0011111",

    "1+D" : "0011111",

    "A+1" : "0110111",

    "1+A" : "0110111",

    "M+1" : "1110111",

    "1+M" : "1110111",

    "D-1" : "0001110",

    "A-1" : "0110010",

    "M-1" : "1110010",

    "D+A" : "0000010",

    "A+D" : "0000010",

    "D+M" : "1000010",

    "M+D" : "1000010",

    "D-A" : "0010011",

    "A-D" : "0000111",

    "D-M" : "1010011",

    "M-D" : "1000111",

    "D&A" : "0000000",

    "D&M" : "1000000",

    "D|A" : "0010101",

    "D|M" : "1010101",

    "A&D" : "0000000",

    "M&D" : "1000000",

    "A|D" : "0010101",

    "M|D" : "1010101"

    }

#Defining the values of destination instructions

destination\_dictionary = {

    'null':  '000',

    'M':   '001',

    'D':   '010',

    'MD':  '011',

    'DM':  '011',

    'A':   '100',

    'AM':  '101',

    'MA':  '101',

    'AD':  '110',

    'DA':  '110',

    'AMD': '111',

    'DMA': '111',

    'ADM': '111',

    'MAD': '111',

    'MDA': '111',

    'DAM': '111',

    }

#Defining the values of jump instructions

jump\_dictionary = {

    'null':  '000',

    'JGT': '001',

    'JEQ': '010',

    'JGE': '011',

    'JLT': '100',

    'JNE': '101',

    'JLE': '110',

    "JMP": '111',

    }

#Defining the values of different symbols

symbols = {

    'R0':'0000000000000000',

    'R1':'0000000000000001',

    'R2':'0000000000000010',

    'R3':'0000000000000011',

    'R4':'0000000000000100',

    'R5':'0000000000000101',

    'R6':'0000000000000110',

    'R7':'0000000000000111',

    'R8':'0000000000001000',

    'R9':'0000000000001001',

    'R10':'0000000000001010',

    'R11':'0000000000001011',

    'R12':'0000000000001100',

    'R13':'0000000000001101',

    'R14':'0000000000001110',

    'R15':'0000000000001111',

    'KBD':'0110000000000000',       #16384 in binary

    'SCREEN':'0100000000000000',    #24576 in binary

    'SP':'0000000000000000',

    'LCL':'0000000000000001',

    'ARG':'0000000000000010',

    'THIS':'0000000000000011',

    'THAT':'0000000000000100'

}

z=[]        #empty list to store the binary values

# Define a dictionary to store the labels and their corresponding addresses

labels = {}

variable={}

lst2=(list(range(16,16384)))

# Open the input file and read the lines

f = open('input.asm', 'r')

lines = f.readlines()

print(lines)

#White spaces and comments identification and deletion

code=[s.replace('\n','') for s in lines]

code=[s.replace('\t','') for s in code]

code=[s.split("\\")[0] for s in code]

code=[s.split("//")[0] for s in code]

code=[s.replace(" ","")for s in code]

code =[x for x in code if x!='']

#Checking labels and adding into symbols table

print(code)

current\_line = 0

label\_dict = {}

# Check if this is a label

for line in code:

    if line[0] == '(' :         # Get the label name and store its line number in the label dictionary

        l=line[1:-1]

        label\_dict[l] = current\_line

    else:

        current\_line += 1

# A instruction

for line in code:

    if line[0] == "(":

        pass

    elif line[0] == "@":

        if line[1:].isdigit():

            # This is an A instruction with a constant value

            value = int(line[1:].strip())

            binary\_value=f'{value:016b}'          # converting value after @ into a 16-bit binary value

            z.append(binary\_value)                # appending the binary value into z[]

        elif line[1:] in symbols.keys():    #checking if it is a symbol in the symbol dictionary

            symbol = line[1:].strip()

            y=symbols[symbol]

            z.append(y)

        elif line[1:] in label\_dict.keys():    #checking if it is a symbol in the symbol dictionary

            label = line[1:].strip()

            y=label\_dict[label]

            z.append( f'{y:016b}')

        else:

            if line[1:].strip() not in variable.keys():

                y=lst2.pop(0)

                variable[line[1:]] = y   # Handle symbols that have not yet been defined (i.e., variables)

                binary\_value = f'{y:016b}'

                z.append(binary\_value)

            else:

                y=variable[line[1:]]

                z.append(f'{y:016b}')

#Cinstruction

    else:

        s=line.split('=')

        c=[]

        if len(s)==1:

            s=line.split(';')

            c=('111'+computation\_dictionary[s[0]]+destination\_dictionary['null']+jump\_dictionary[s[1]])

            z.append(c)

        else:

            t=s[1].split(';')

            if len(t)==2:

                c=('111'+computation\_dictionary[t[0]]+destination\_dictionary [s[0]]+jump\_dictionary[t[1]])

                z.append(c)

            else:

                c='111'+computation\_dictionary[t[0]]+destination\_dictionary [s[0]]+jump\_dictionary['null']

                z.append(c)

print(z)

f2 = open('output.hack', 'w')

for j in z:

    f2.writelines((j)+"\n")

f.close()

f2.close()

# 

# CODE EXPLANATION

computation\_dictionary = {

   . . .

    }

This creates a dictionary called **computation\_dictionary** that maps computation instructions to their corresponding binary values.

destination\_dictionary = {

   . . .

    }

This creates a dictionary called **destination\_dictionary** that maps destination instructions to their corresponding binary values.

jump\_dictionary = {

   . . .

    }

This creates a dictionary called **jump\_dictionary** that maps jump instructions to their corresponding binary values.

symbols = {

   . . .

    }

This creates a dictionary called **symbols** that maps symbols to their corresponding binary values.

z=[]

This initializes an empty list called **z** that will be used to store the binary values of each line of machine code.

labels = {}

variable={}

lst2=(list(range(16,16384)))

This initializes two empty dictionaries called **labels** and **variable**, and a list called **lst2** that contains all the memory locations available for variables (excluding the first 16 which are reserved for specific purposes).

f = open('input.asm', 'r')

lines = f.readlines()

This opens the file called **input.asm** in read mode and reads all the lines of the file into a list called **lines**.

code=[s.replace('\n','') for s in lines]

code=[s.replace('\t','') for s in code]

code=[s.split("\\")[0] for s in code]

code=[s.split("//")[0] for s in code]

code=[s.replace(" ","")for s in code]

code =[x for x in code if x!='']

This cleans up the input code by removing whitespace, newlines, and comments. The code is then split into a list called **code** where each item in the list represents a line of code.

print(code)

current\_line = 0

label\_dict = {}

for line in code:

    if line[0] == '(' :

        l=line[1:-1]

        label\_dict[l] = current\_line

    else:

        current\_line += 1

This checks for labels in the code and adds them to a dictionary called **label\_dict**. If a line of code starts with a **(**character, the code between the parentheses is extracted and used as the key in the dictionary. The value associated with the key is the current line number (which will be used later to replace labels with their corresponding memory addresses). If the line of code doesn't start with a **(**character, then the **current\_line** counter is incremented by 1.

**A-Instruction:**

for line in code:

This line initializes a loop that will iterate over each line of code passed in as an argument to the function.

if line[0] == "(":

        pass

This line checks whether the first character of the current line is an opening parenthesis. If it is, the code "passes" over that line (i.e., does nothing).

elif line[0] == "@":

If the first character of the line is not an opening parenthesis, this line checks whether it is an at sign. If it is, the following block of code is executed.

if line[1:].isdigit():

            value = int(line[1:].strip())

            binary\_value=f'{value:016b}'

            z.append(binary\_value)

This block of code checks whether the characters after the at sign form a number. If they do, it converts that number to an integer, converts that integer to a 16-bit binary value, and appends the binary value to a list called **z**.

elif line[1:] in symbols.keys():

            symbol = line[1:].strip()

            y=symbols[symbol]

            z.append(y)

This block of code checks whether the characters after the at sign form a symbol that is in a dictionary called **symbols**. If they do, it looks up the value of that symbol in the **symbols** dictionary and appends that value to **z**.

elif line[1:] in label\_dict.keys():

            label = line[1:].strip()

            y=label\_dict[label]

            z.append( f'{y:016b}')

This block of code checks whether the characters after the at sign form a label that has already been defined in a dictionary called **label\_dict**. If they do, it gets the value of that label (which should be an integer), converts it to a 16-bit binary value, and appends that binary value to **z**.

else:

            if line[1:].strip() not in variable.keys():

                y=lst2.pop(0)

                variable[line[1:]] = y   # Handle symbols that have not yet been defined (i.e., variables)

                binary\_value = f'{y:016b}'

                z.append(binary\_value)

            else:

                y=variable[line[1:]]

                z.append(f'{y:016b}')

If the characters after the at sign do not match any of the above conditions, this block of code checks whether the symbol is already in a dictionary called **variable**. If it is, it gets the value of that symbol from **variable**, converts it to a 16-bit binary value, and appends that binary value to **z**. If the symbol is not in **variable**, this code assigns the next available value from a list called **lst2** to the symbol, adds the symbol to **variable** with its corresponding value, converts that value to a 16-bit binary value, and appends that binary value to **z**.

**C-Instruction:**

else:

    s=line.split('=')

If the instruction does not start with ‘@’, then it executes the code for C-Instruction by first splitting the string stored in the variable 'line' at every occurrence of the '=' character and stores the resulting substrings in a list called 's'.

 c=[]

This line creates an empty list called 'c'.

    if len(s)==1:

This line checks if the length of the list 's' is equal to 1. If it is, then this indicates that there was no '=' character in the original string, and therefore it must be split at the ';' character instead.

s=line.split(';')

This line splits the string stored in the variable 'line' at every occurrence of the ';' character and stores the resulting substrings in a list called 's'.

c=('111'+computation\_dictionary[s[0]]+destination\_dictionary['null']+jump\_dictionary[s[1]])

z.append(c)

This line creates a new string by concatenating the values stored in three dictionaries with the appropriate keys based on the substrings in the 's' list. The resulting string is stored in the 'c' variable. The string in ‘c’ is then appended into ‘z’.

else:

     t=s[1].split(';')

If the length of the 's' list is not equal to 1 (i.e., the original string contained the '=' character), then the line splits the second substring from the 's' list at every occurrence of the ';' character and stores the resulting substrings in a list called 't'.

if len(t)==2:

c=('111'+computation\_dictionary[t[0]]+destination\_dictionary[s[0]]+jump\_dictionary[t[1]])

z.append(c)

This line checks if the length of the 't' list is equal to 2. If it is, then this indicates that there are two substrings in the 't' list which is the computational and the jump bits. A new string is then created by concatenating the values stored in the dictionaries with the appropriate keys based on the substrings in the 't' and 's' lists. The resulting string is stored in the 'c' variable. The value in ‘c’ is then appended in ‘z’.

else:

c='111'+computation\_dictionary[t[0]]+destination\_dictionary[s[0]]+jump\_dictionary['null']

z.append(c)

If the length of the 't' list is not equal to 2, i.e., the instruction contains only C-Instruction, then it constructs a new string containing only the computational and destination instructions and stores the concatenated string in a variable ‘c’. the value in ‘c’ is then appended in ‘z’.

print(z)

This prints the result stored in ‘z’.

f2 = open('output.hack', 'w')

for j in z:

    f2.writelines((j)+"\n")

f.close()

f2.close()

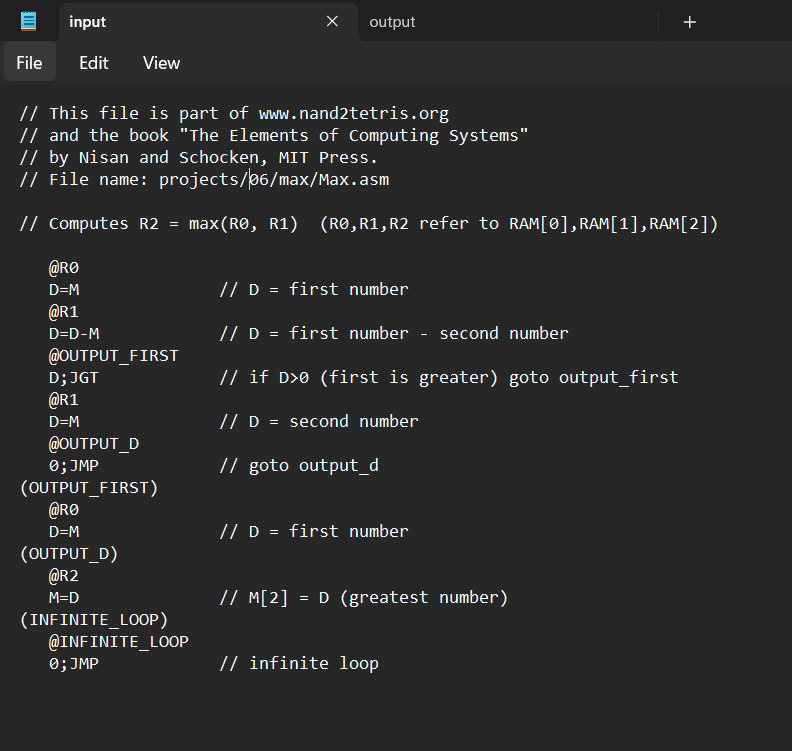
This opens the output.hack file and writes the output ‘z’ which contains the binary representation of the input asm file into it.

The input and output files are then closed.

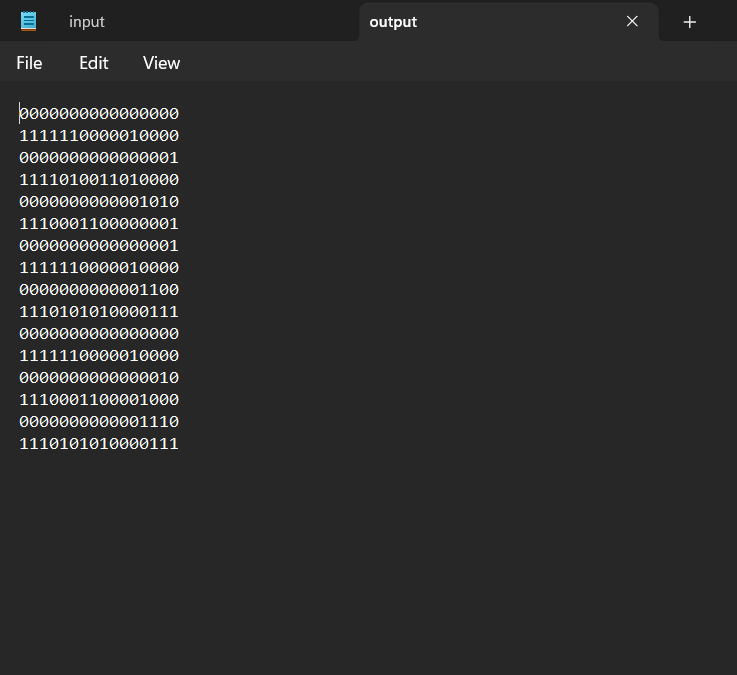
# RESULT

Converting Max from assembly language to binary:

**Input file:**

****

**Output file:**

****

Comparing the built assembler result with HACK Assembler result:

