Table of Contents

ABSTRACT .......................................................................................................... 5

BUILDING OF THE VM TRANSLATOR ......................................................... 6

INTRODUCTION ............................................................................................. 6

*Calling Commands*: ........................................................................................ 6

*Memory Access Commands*: ......................................................................... 11

*Arithmetic Commands*: ................................................................................. 16

*Program Flow Commands* ............................................................................ 21

CODING THE VM TRANSLATOR USING PYTHON ................................... 22

RESULT .............................................................................................................. 37

References ........................................................................................................... 39

LIST OF FIGURES

*Figure 1.Calling Procedure ------------------------------------------------------------- 6*

*Figure 2.Function Procedure ----------------------------------------------------------- 8*

*Figure 3.Return Procedure ------------------------------------------------------------- 8*

*Figure 4. Input VM code -------------------------------------------------------------- 37*

*Figure 5. Output ASM code ----------------------------------------------------------- 38*

ABSTRACT

The VM Translator helps in bridging the gap between high-level machine language and the Hardware component. It is the next step to building an assembler and helps convert machine language into assembly language.

Virtual machines have become increasingly prevalent in the field of software development, enabling programmers to write code in higher-level languages while maintaining compatibility across different hardware platforms. The VM Translator serves as a crucial intermediary between the high-level code written by programmers and the low-level instructions executed by the underlying hardware.

The primary objective of the VM Translator is to translate platform-independent, high-level code into the specific assembly language instructions understood by the target hardware. This translation process involves various stages, including lexical analysis, parsing, semantic analysis, code generation, and optimization techniques. By performing these steps, the VM Translator ensures efficient and accurate execution of the high-level code on the target hardware.

The high- levelled language like Jack is first converted into an intermediate stage of VM language before it gets converted into assembly language and further into binary language.

In this project, we convert this VM language into assembly language with the help of a VM translator, which is built using python.

# BUILDING OF THE VM TRANSLATOR

## INTRODUCTION

The VM Translator built here has mainly four types of commands- arithmetic, memory access, program flow, and calling commands. The VM Translator is built in python and provides an insight on the major possible functions.

The operands and results of the VM operations resides in a stack data structure. In a stack machine model, arithmetic commands pop their operands from the top of the stack and push the results back onto the stack. There is also a possible transfer of data items from the stacks top to the designated memory locations. Any program can be translated into an equivalent stack machine program. A stack is an abstract data type consisting of two basic operations- push and pop. The push operation adds an element on the top of the stack, above the previously pushed element. The pop operation retrieves and removes the top element. The stack follows the LIFO rule (Last In First Out). The stack data structure can be implemented by keeping a stack and a stack pointer variable, *sp*, that points to the available location above the topmost element. The *push x* command is implemented by storing x t the array entry pointed to by the sp and then incrementing sp. The *pop x* is implemented by first decrementing the stack pointer sp and then returning the value stored in the topmost position.

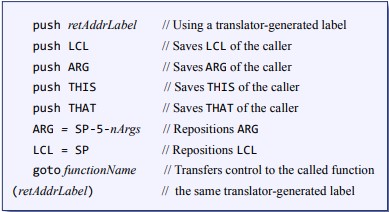
The building of the VM Translator involves the implementation of 5 functions- call, function, labels, arithmetic functions and the return function.

### Calling Commands**:**

(a) CALLING: The VM command of handling a call has the following syntax-

**Call *functionName* nArgs**

It calls a function, informing nArgs arguments have been pushed onto the stack. In the assembly code aspect, the pseudocode of the callfunction is as follows: -



*Figure 1.Calling Procedure*

* Save return address

|  |
| --- |
| @return\_address  D=A  @SP  A=M  M=D  @SP  M=M+1 |

* Save LCL, ARG, THIS and THAT

|  |
| --- |
| @SP  D=M  @LCL  M=D |

* Set LCL

|  |
| --- |
| @SP  D=M  @LCL  M=D |

* Set ARGS

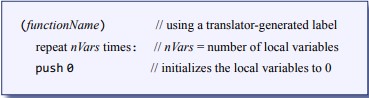
|  |
| --- |
| @SP  D=M  @5  D=D-A  @num\_args  D=D-A  @ARG  M=D |

* Jump to called function

|  |
| --- |
| @function\_name  0;JMP |

1. FUNCTION: The VM command of handling a function has the following syntax-

**Function *functionName nVars***

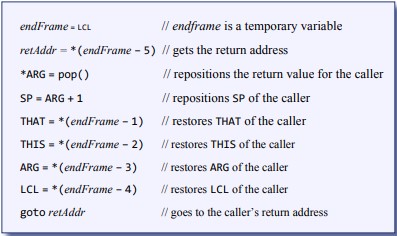
Gives the location where the start of a function that has nVars local variables.

*Figure 2.Function Procedure*

|  |
| --- |
| @function\_name  D=A  @SP  A=M  M=D  @SP  M=M+1 |

1. RETURN: The VM command of handling a return has the following syntax-

**Return**

 *Figure 3.Return Procedure*

* Setting endFrame in a temporary variable

|  |
| --- |
| @LCL  D=M  @R13  M=D |

* Getting the return address to LCL-5 and storing in LCL-5

|  |
| --- |
| @5  A=D-A  D=M  @R14  M=D |

* Storing return value in Arg 0.

|  |
| --- |
| @SP  A=M-1  D=M  @ARG  A=M  M=D |

* Set SP to Arg 1

|  |
| --- |
| D=A+1  @SP  M=D |

* Restoring the THAT pointer to (frame-1)

|  |
| --- |
| @R13  AM=M-1  D=M  @THAT  M=D |

* Restoring the THIS pointer to (frame-2)

|  |
| --- |
| @R13  AM=M-1  D=M  @THIS  M=D |

* Restoring the Args pointer to (frame-3)

|  |
| --- |
| @R13  AM=M-1  D=M  @ARG  M=D |

* Restoring LCL pointer to (frame-4)

|  |
| --- |
| @R13  AM=M-1  D=M  @LCL  M=D |

* Going to the retAddress

|  |
| --- |
| @R14  A=M  0;JMP |

### Memory Access Commands**:**

All the memory segments are accessed by two commands- the push command and the pop command.

1. Local: It stores the function’s local variables. It is allocated dynamically by the VM implementation and initialized to 0 when the function is entered

* 1. Push Local i **addr = LCL+I, \*sp = \*addr, sp++**

|  |
| --- |
| @256  D=A  @sp  M=D  @LCL  D=M  @i  A=D+A  D=M  @SP  A=M  M=D  @SP  M=M+1 |

* 1. Pop Local i **addr = LCL+I, SP--, \*addr=\*sp**

|  |
| --- |
| @256  D=A  @sp  M=D  @LCL  D=M  @i  D=D+A  @addr M=D  @addr  A=M  M=D |

1. Constant: Pseudo segments that holds all the constant values in the range of 0 to 32767.
   1. Push Constant i

**\*sp = I, sp++**

|  |
| --- |
| @256  D=A  @sp  M=D  @i  D=A  @SP  A=M  M=D  @SP  M=M+1 |

1. Temp: Fixed eight-entry segment from 5 to 12 that holds temporary variables for general use.

* 1. Push Temp i

**addr = 5 + I, \*sp=\*addr, sp++**

|  |
| --- |
| @256  D=A  @sp  M=D  @(5+i)  D=M  @SP  A=M  M=D  @SP  M=M+1 |

* 1. Pop Temp i

**addr = 5+I, sp--, \*addr = \*sp**

|  |
| --- |
| @256  D=A  @sp  M=D  @SP  AM=M-1  D=M  @(5+i)  M=D  @addr  A=M  M=D |

1. Argument: Stores the function’s arguments which is dynamically allocated by the VM implementation when a function is entered.

This/ That: General purpose segments that are used to manipulate selected areas in a heap.

* 1. Push Segment I (Segment: - Argument, This, That)

**addr = SegmentPointer + I, \*sp=\*addr, sp++**

|  |
| --- |
| @256  D=A  @sp  M=D  @(Segment)  D=M  @i  A=D+A  D=M  @SP  A=M  M=D  @SP  M=M+1 |

* 1. Pop Segment I (Segment: - Argument, This, That) **addr = SegmentPointer + I, sp--, \*addr = \*sp**

|  |
| --- |
| @256  D=A  @sp  M=D  @(Segment)  D=M  @i  D=D+A  @addr M=D  @addr  A=M  M=D |

1. Pointer: A two-entry segment that holds the base address for this and that.

* 1. Push Pointer 0/1

**\*sp = THIS/THAT, sp++**

|  |
| --- |
| "@(THIS/THAT)"  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1" |

* 1. Pop Pointer 0/1

**sp--, THIS/THAT=\*sp**

|  |
| --- |
| "@SP",  "AM=M-1",  "D=M",  "@THIS/THAT(index)"  "M=D" |

1. Static: Stored the static variables shared by all functions in the same .vm file.
   1. Push static i

|  |
| --- |
| "@function\_index"  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1" |

* 1. Pop static i

|  |
| --- |
| "@function.index",  "D=A",  "@13",  "M=D",  "@SP",  "AM=M-1",  "D=M",  "@R13",  "A=M",  "M=D" |

### Arithmetic Commands**:**

performs arithmetic and logical operations on the stack. The basic arithmetic functions of add, subtract, neg, or, and, not, greater-than, lesser-than, equal-to are implemented in the VM Translator by writing its corresponding assembly language.

1. ADD:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  M=D+M |

First, we set the A-register to the address of the stack pointer (SP). The stack pointer points to the topmost element on the stack in Hack assembly. This line decreases the value of the A-register and M-register (memory) by 1. It moves the A-register to the address of the topmost element on the stack and decrements the stack pointer. Then, the value stored at the address pointed to by the A-register is copied into the D-register. The value is temporarily stored in the D-register for later use. Then we decrement the value of the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. Then, we perform the addition operation. It adds the value stored in the D-register to the value stored at the address pointed to by the A-register. The result of the addition is then stored back into the address pointed to by the A-register, effectively replacingthe second-to-topmost element with the sum.

1. SUBTRACT:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  M=M-D |

Similar to the addition operation, we set the A-register to the address of the stack pointer (SP). The stack pointer points to the topmost element on the stack in Hack assembly. This line decreases the value of the A-register and M-register (memory) by 1. It moves the A-register to the address of the topmost element on the stack and decrements the stack pointer. Then, the value stored at the address pointed to by the A-register is copied into the D-register. The value is temporarily stored in the D-register for later use. Then we decrement the value of the Aregister by 1, effectively moving the A-register to the address of the second-totopmost element on the stack. Then, we perform the subtraction operation. It subtracts the value stored in the D-register to the value stored at the address pointed to by the A-register. The result of the subtraction is then stored back into the address pointed to by the A-register, effectively replacingthe second-totopmost element with the difference.

1. NEGATION:

|  |
| --- |
| @SP  A=M-1  M=-M |

This first line sets the A-register to the address of the stack pointer (SP). The stack pointer points to the topmost element on the stack in Hack assembly.Then, in the next line decrements the value of the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. The value stored at the address pointed to by the A-register is negated, which is the second-to-topmost element on the stack. Similar to the previous negation operation, it multiplies the value by -1 and stores the negated value back into the same memory location.

1. EQUAL-TO:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  D=M-D  @EQ\_TRUE  D;JEQ @SP  A=M-1  M=0  @EQ\_END  0;JMP  (EQ\_TRUE)  @SP  A=M-1  M=-1  (EQ\_END) |

First, the stack pointer is decremented to point to the second-to-topmost element, and the topmost element is stored in the D-register for later use. Then, we decrement the address in the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. Then, it subtracts the value stored in the D-register from the value at the address pointed to by the

A-register. The result of the subtraction is stored back into the D-register. Then, we check if the subtraction result is equal to zero. If the result is zero, the code jumps to the label "EQ\_TRUE" using the JEQ (Jump if Equal) instruction. This line sets the A-register to the address of the topmost element on the stack. Then, it stores the value 0 at that memory location, indicating that the elements on the stack are not equal, it defines the label "EQ\_END" and unconditionally jump to it using the JMP (Jump) instruction. This ensures that the code execution continues past the block responsible for setting the value 0 when the elements are not equal. Then we define the label "EQ\_TRUE" and set the A-register to the address of the topmost element on the stack. Then, it stores the value -1 (representing true or equality) at that memory location.

1. LESSER-THAN:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  D=M-D  @LT\_TRUE  D;JLT @SP  A=M-1  M=0  @LT\_END  0;JMP  (LT\_TRUE)  @SP  A=M-1  M=-1  (LT\_END) |

First, we decrement the stack pointer (SP) and store the topmost element of the stack into the D-register. The stack pointer is decremented to point to the secondto-topmost element, and the topmost element is stored in the D-register for later use. This line decrements the address in the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. Then, it subtracts the value stored in the D-register from the value at the address pointed to by the A-register. The result of the subtraction is stored back into the Dregister. These lines check if the subtraction result is less than zero. If the result is less than zero, the code jumps to the label "LT\_TRUE" using the JLT instruction. Then we set the A-register to the address of the topmost element on the stack. Then, it stores the value 0 at that memory location, indicating that the elements on the stack do not satisfy the less-than condition. These lines define the label "LT\_END" and unconditionally jump to it using the JMP instruction. This ensures that the code execution continues past the block responsible for setting the value 0 when the condition is not satisfied. These lines define the label "LT\_TRUE" and set the A-register to the address of the topmost element on the stack. Then, it stores the value -1 at that memory location.

1. GREATER-THAN:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  D=M-D  @GT\_TRUE  D;JGT @SP  A=M-1  M=0  @GT\_END  0;JMP  (GT\_TRUE)  @SP  A=M-1  M=-1  (GT\_END) |

First, we decrement the stack pointer (SP) and store the topmost element of the stack into the D-register. The stack pointer is decremented to point to the secondto-topmost element, and the topmost element is stored in the D-register for later use. This line decrements the address in the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. Then, it subtracts the value stored in the D-register from the value at the address pointed to by the A-register. The result of the subtraction is stored back into the Dregister. These lines check if the subtraction result is greater than zero. If the result is greater than zero, the code jumps to the label "GT\_TRUE" using the JGT instruction. This line sets the A-register to the address of the topmost element on the stack. Then, it stores the value 0 at that memory location, indicating that the elements on the stack do not satisfy the greater-than condition. These lines define the label "GT\_END" and unconditionally jump to it using the JMP instruction. This ensures that the code execution continues past the block responsible for setting the value 0 when the condition is not satisfied. These lines define the label "GT\_TRUE" and set the A-register to the address of the topmost element on the stack. Then, it stores the value -1 at that memory location.

1. AND:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  M=D&M |

First, we decrement the stack pointer (SP) and store the topmost element of the stack into the D-register. The stack pointer is then decremented to point to the second-to-topmost element, and the topmost element is stored in the Dregister for later use. We then decrement the address in the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. Then, it performs the bitwise AND operation between the value stored in the D-register and the value at the address pointed to by the A-register. The result of the AND operation is stored back into the memory location of the second-to-topmost element.

1. OR:

|  |
| --- |
| @SP  AM=M-1  D=M  A=A-1  M=D|M |

Similar to the and operation, we first decrement the stack pointer (SP) and store the topmost element of the stack into the D-register. The stack pointer is then decremented to point to the second-to-topmost element, and the topmost element is stored in the D-register for later use. We then decrement the address in the A-register by 1, effectively moving the A-register to the address of the second-to-topmost element on the stack. Then, it performs the bitwise OR operation between the value stored in the D-register and the value at the address pointed to by the A-register. The result of the OR operation is stored back into the memory location of the second-to-topmost element.

1. NOT

|  |  |
| --- | --- |
|  | |
|  | @SP  A=M-1  M=!M |
|  |

We first set the A-register to the address of the topmost element on the stack. The stack pointer (SP) points to the topmost element, and the A-register is set to that address. Then we perform the bitwise logical NOT operation on the value stored at the address pointed to by the A-register. It complements each bit of the value, toggling 1s to 0s and 0s to 1s. The result of the logical NOT operation is stored back into the same memory location, effectively replacing the topmost element with its logical complement.

### Program Flow Commands

Program flow commands in Hack assembly language are used to control the flow of execution within a program

1. LABEL: Labels are instruction addresses that allow jumps in the program easier to handle. It just contains a name for the label that creates a bifurcation in the code for easy jumping.

1. goto-symbol: It performs a one-way transfer of control to another line of code. On encountering a *@label,* it jumps to that location

|  |
| --- |
| "@"+label,  "0;JMP" |

1. If-goto symbol: It performs a one-way transfer of control to another line of code based on certain constrains.

|  |
| --- |
| "@SP",  "AM=M-1",  "D=M",  "@"+label,  "D;JNE" |

|  |
| --- |
| CODING THE VM TRANSLATOR USING PYTHON    # Define the VM translator function def translate\_vm(commands):  # Define a dictionary to map VM segment names to their corresponding memory addresses  segment\_map = { "argument": "ARG",  "local": "LCL",  "static": "16",  "this": "THIS",  "that": "THAT",  "temp": "5",  "SP": "256"  } name="input" filename="input.vm" call\_counter = 0 func="" pointer\_map ={"0":"THIS", "1":"THAT"} # Open the input file and read the lines f = open('input.vm', 'r') lines = f.readlines() print(lines)    #White spaces and comments identification and deletion lines=[s.replace('\n','') for s in lines] lines=[s.replace('\t','') for s in lines] lines=[s.split("\\")[0] for s in lines] lines=[s.split("//")[0] for s in lines] lines=[s.strip() for s in lines] lines = [x for x in lines if x != ''] lines=[s.split(" ") for s in lines] dir\_path = os.path.dirname(os.path.realpath(filename)) dir\_list = os.listdir(dir\_path) dir\_list.remove(filename)  print(dir\_path) print(dir\_list) print(lines) c=[] fun=[] fc=0 for i in lines: if (i[0]=="function"):  fc+=1 elif (i[0]=="call"): fc+=1  22 | P a g e |

fun.append(i[1].split(".")[0])

c.append(i[1].split(".")[1]) for j in dir\_list: print(j.split("."))

|  |
| --- |
| if j.split(".")[0] in fun: if j.split(".")[1] =="vm": f2=open(j,'r') sentence=f2.readlines() sentence=[s.replace('\n','') for s in sentence] sentence=[s.replace('\t','') for s in sentence] sentence=[s.split("\\")[0] for s in sentence] sentence=[s.split("//")[0] for s in sentence] sentence=[s.strip() for s in sentence] sentence = [x for x in sentence if x != ''] sentence=[s.split(" ") for s in sentence] print(sentence) lines=sentence+lines |

Create a segment map and enter the predefined values and the mapping of the elements. The *pointer\_map* stores the THIS and THAT values.

Open the “*input.vm”* file to enter the VM code that needs to be converted to the asm format.

Remove the white spaces, inline and line comments and the tab spaces present in the VM file before converting it into the asm file.

Create an empty list “*output\_lines”* to store the converted lines of the VM file. To call functions from an external file, create a *dir\_path,* which stores the path directory and appends the files available in that path into a *dir\_list.* Store the function and the file name so obtained in two separate lists. If the function is present in an external file, then remove the white spaces and comments in that file also before processing.

|  |
| --- |
| output\_lines.append("@256"), output\_lines.append("D=A"), output\_lines.append("@0"), output\_lines.append("M=D") |

We append the above lines to the starting of the asm file. This sets the value of the stack pointer to 256; from where the stack starts.

def call(function\_name, num\_args):

global call\_counter global output\_line # Push return address label

return\_address = "RETURN\_ADDRESS"+function\_name # Use a translator-

|  |
| --- |
| generated label assembly = ["@" + return\_address+str(call\_counter), "D=A", "@SP", "A=M", "M=D", "@SP", "M=M+1"]    # Push LCL, ARG, THIS, and THAT of the caller segment\_pointers = ["LCL", "ARG", "THIS", "THAT"] for pointer in segment\_pointers:  assembly.extend(["@" + pointer, "D=M", "@SP", "A=M", "M=D", "@SP", "M=M+1"])    # Reposition LCL assembly.extend(["@SP", "D=M", "@LCL", "M=D"])    # Reposition ARG  assembly.extend(["@SP", "D=M", "@5", "D=D-A", "@" + str(num\_args), "D=DA", "@ARG", "M=D"])    # Transfer control to the called function assembly.extend(["@" + function\_name, "0;JMP"])    # Return address label assembly.append("(" + return\_address+str(call\_counter) + ")") output\_lines.extend(assembly)    #call counter call\_counter = call\_counter+1 |

The above segment of the code is used to implement the call of the VM code.

#push/pop for all the stacks def write\_push(segment, index): global name global output\_line global pointer\_map

|  |
| --- |
| global func  """  Writes the assembly lines that implements the "push segment index" command in the VM language.  """  # Determine the memory segment to push from. if segment == "constant":  assembly = [  "@"+index,  "D=A",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] elif segment == "local":  assembly = [ "@LCL",  "D=M",  "@"+index,  "A=D+A",  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] elif segment == "argument":  assembly = [ "@ARG", "D=M", f"@{index}",  "A=D+A",  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] elif segment == "this": |

assembly = [ "@THIS", "D=M", f"@{index}", "A=D+A",

|  |
| --- |
| "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] elif segment == "that":  assembly = [ "@THAT", "D=M", f"@{index}",  "A=D+A",  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] elif segment == "pointer":  assembly = [  "@"+pointer\_map[index],  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] elif segment == "temp":  assembly = [  "@"+index,  "D=A",  "@5",  "A=D+A",  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1" |

|  |
| --- |
| ] elif segment == "static": assembly = [  "@"+func+"."+index,  "D=M",  "@SP",  "A=M",  "M=D",  "@SP",  "M=M+1"  ] else:  raise ValueError(f"Invalid segment: {segment}") output\_lines.extend(assembly)    def write\_pop(segment, index):  global output\_lines global pointer\_map global name global func  """  Writes the assembly code that implements the "pop segment index" command in the VM language.  """  # Determine the memory segment to pop to. if segment == "local":  assembly = [ "@LCL", "D=M", f"@{index}",  "D=D+A",  "@R13",  "M=D",  "@SP",  "AM=M-1",  "D=M",  "@R13",  "A=M",  "M=D"  ] elif segment == "argument":  assembly = [ "@ARG", "D=M", f"@{index}",  "D=D+A",  27 | P a g e |

"@R13",

"M=D",

"@SP",

"AM=M-1",

"D=M",

|  |
| --- |
| "@R13",  "A=M",  "M=D"  ] elif segment == "this":  assembly = [ "@THIS", "D=M", f"@{index}",  "D=D+A",  "@R13",  "M=D",  "@SP",  "AM=M-1",  "D=M",  "@R13",  "A=M",  "M=D"  ] elif segment == "that":  assembly = [ "@THAT", "D=M", f"@{index}",  "D=D+A",  "@R13",  "M=D",  "@SP",  "AM=M-1",  "D=M",  "@R13",  "A=M",  "M=D"  ] elif segment == "pointer":  assembly = [ "@SP",  "AM=M-1",  "D=M",  "@"+pointer\_map[index],  "M=D"  ] elif segment == "temp": |

assembly = [ "@"+index,

"D=A",

"@5",

"D=D+A",

|  |
| --- |
| "@13",  "M=D",  "@SP",  "M=M-1",  "A=M",  "D=M",  "@13",  "A=M",  "M=D" ] elif segment == "static": assembly = [  "@"+func+"."+index,  "D=A",  "@13",  "M=D",  "@SP",  "AM=M-1",  "D=M",  "@R13",  "A=M",  "M=D"  ] else:  raise ValueError(f"Invalid segment: {segment}") output\_lines.extend(assembly) |

The above lines of the code are used to implement the push and pop memory access functions. We then append the assembly lines to the “*output\_lines”* which helps print the compiled output on the later stages.

|  |
| --- |
| if (fc>1):  call("Sys.init",0) |

If there exists more than 1 function in the VM code, then, we call and execute the main function, which here is *Sys.init.*

#If-goto and goto def if\_goto(label): global output\_lines assembly = [

"@SP",

|  |
| --- |
| "AM=M-1",  "D=M",  "@"+label,  "D;JNE" ] output\_lines.extend(assembly)  def goto(label): global output\_lines assembly = [ "@"+label,  "0;JMP"  ] output\_lines.extend(assembly)  def label(label):  global output\_lines output\_lines.append("("+label+")") |

The above lines of the code are used to implement the if and if-goto functions of the VM code and also deals with the label implementation and compiles it into the assembly language.

|  |
| --- |
| #functions implementation: def functions(f\_name,n\_vars):  global output\_lines inst=["("+f\_name+")"] for i in range(int(n\_vars)):  inst.extend([ "@SP",  "A=M",  "M=0",  "@SP",  "M=M+1"  ]) output\_lines.extend(inst) |

The above code deals with the function implementation.

|  |
| --- |
| def arithmetic(command): global output\_lines if command == "add":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("M=D+M") elif command == "sub":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("M=M-D") elif command == "neg":  output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=-M") elif command == "eq":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("D=M-D") output\_lines.append("@EQ\_TRUE") output\_lines.append("D;JEQ") output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=0") output\_lines.append("@EQ\_END") output\_lines.append("0;JMP") output\_lines.append("(EQ\_TRUE)") output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=-1") output\_lines.append("(EQ\_END)") elif command == "gt":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("D=M-D") output\_lines.append("@GT\_TRUE") output\_lines.append("D;JGT") output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=0")  31 | P a g e |

output\_lines.append("@GT\_END") output\_lines.append("0;JMP") output\_lines.append("(GT\_TRUE)") output\_lines.append("@SP") output\_lines.append("A=M-1")

|  |
| --- |
| output\_lines.append("M=-1") output\_lines.append("(GT\_END)") elif command == "lt":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("D=M-D") output\_lines.append("@LT\_TRUE") output\_lines.append("D;JLT") output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=0") output\_lines.append("@LT\_END") output\_lines.append("0;JMP") output\_lines.append("(LT\_TRUE)") output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=-1") output\_lines.append("(LT\_END)") elif command == "and":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("M=D&M") elif command == "or":  output\_lines.append("@SP") output\_lines.append("AM=M-1") output\_lines.append("D=M") output\_lines.append("A=A-1") output\_lines.append("M=D|M") elif command == "not":  output\_lines.append("@SP") output\_lines.append("A=M-1") output\_lines.append("M=!M") else:  return "" print(output\_lines) |

The above lines of the code are used to implement the basic arithmetic operations of add. Subtract, negation, or, not, and, greater-than, lesser-than and equal-to.

|  |
| --- |
| def write\_return():  global output\_lines  """  Writes the assembly lines that implements the "return" command in the VM language.  """  # Store the current frame pointer (LCL) in a temporary variable.  # This will be used later to restore the caller's state. assembly = [  "// return",  "@LCL", "D=M",  "@R13",  "M=D"  ]      # Store the return address (the value at the top of the caller's stack) in a temporary variable. assembly += [  # Get the return address (stored at LCL - 5) and store it in R14.  "@5",  "A=D-A",  "D=M",  "@R14",  "M=D"  ]      # Move the return value (the value at the top of the callee's stack) to the caller's stack. assembly += [  # Get the return value (stored at the top of the callee's stack) and store it in ARG 0. "@SP",  "M=M-1",  "A=M",  "D=M",  "@ARG",  "A=M",  "M=D"  ]      # Restore the caller's stack pointer.  assembly += [  # Set SP to ARG + 1  "@ARG",  33 | P a g e |

|  |  |
| --- | --- |
| "D=M",  "@SP",  "M=D+1"  ]      # Restore the caller's state by restoring the caller's segment pointers. assembly += [  # Restore THAT pointer (THAT = \*(frame - 1))  "@R13",  "AM=M-1",  "D=M",  "@THAT",  "M=D",    # Restore THIS pointer (THIS = \*(frame - 2))  "@R13",  "AM=M-1",  "D=M",  "@THIS",  "M=D",    # Restore ARG pointer (ARG = \*(frame - 3))  "@R13",  "AM=M-1",  "D=M",  "@ARG",  "M=D",    # Restore LCL pointer (LCL = \*(frame - 4))  "@R13",  "AM=M-1",  "D=M",  "@LCL",  "M=D",  ]    # Jump to the return address. assembly += [  # Jump to the return address (stored in R14).  "@R14",  "A=M",  "0;JMP"  ]  output\_lines.extend(assembly)  The above code is used to implement the return call of the VM code. This helps to refresh the stack and get the stack pointer to its original positions.    34 | P a g e | |
| for line in lines: segments = line print(segments) if len(segments) == 1:  if segments[0] in ["add", "sub", "neg","eq", "gt", "lt", "and", "or",  "not"]: output\_lines.append(arithmetic(segments[0])) elif segments[0] == "return":  write\_return() else:  output\_lines.append(" ".join(segments)) elif len(segments)==3: print(segments) if segments[0] == "push":  write\_push(segments[1],segments[2]) elif segments[0] == "pop":  write\_pop(segments[1],segments[2]) elif segments[0] == "call":  call(segments[1],segments[2]) elif segments[0] == "function": func=segments[1].split(".")[0] functions(segments[1],segments[2]) elif len(segments) == 2: if segments[0] == "if-goto": if\_goto(segments[1]) elif segments[0] == "goto":  goto(segments[1]) elif segments[0] == "label":  label(segments[1]) |

The above lines of the code is used to parse the lines of the VM code and print the asm code for the corresponding segment.

|  |
| --- |
| output\_lines.append("(End)") output\_lines.append("@End") output\_lines.append("0;JMP") print(output\_lines) |

Append the above lines to the bottom of the assembly code to implement an infinite loop. This ensures that the program does not come through any malicious code after the execution of the actual code.

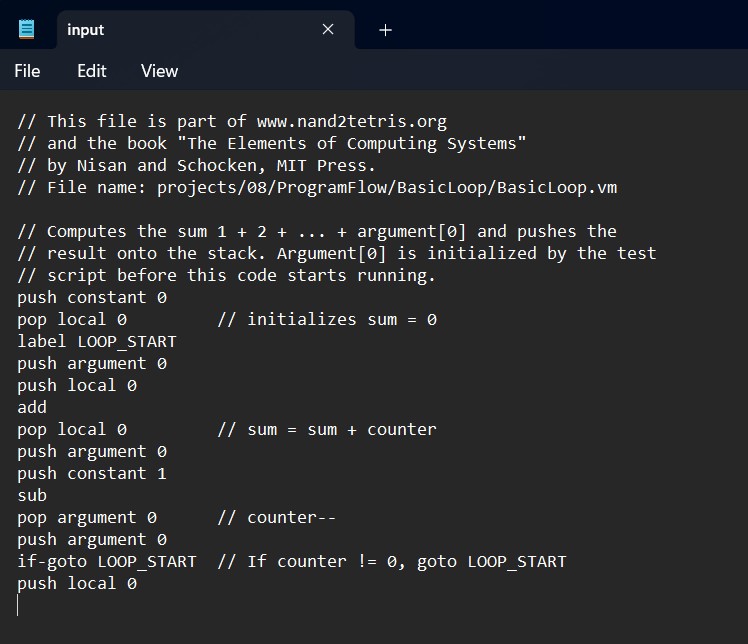
|  |
| --- |
| print(output\_lines) f2 = open('input.asm', 'w') for j in output\_lines: if j== None:  pass else: f2.writelines((j)+"\n")  f.close() f2.close() |

Open a new “*input.asm”* file and write the converted assembly code in it. Then, close the opened VM files.

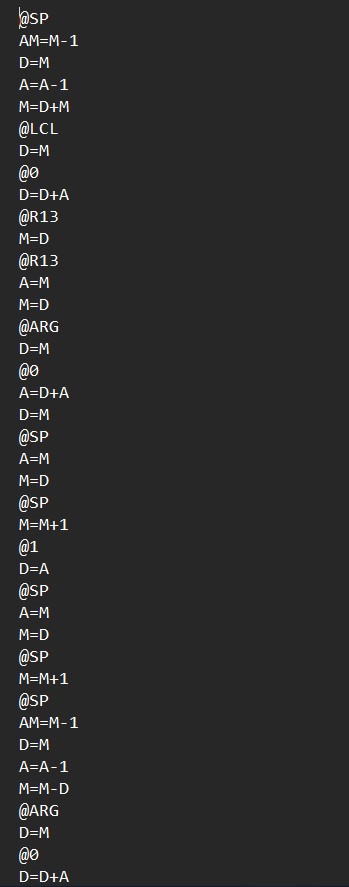
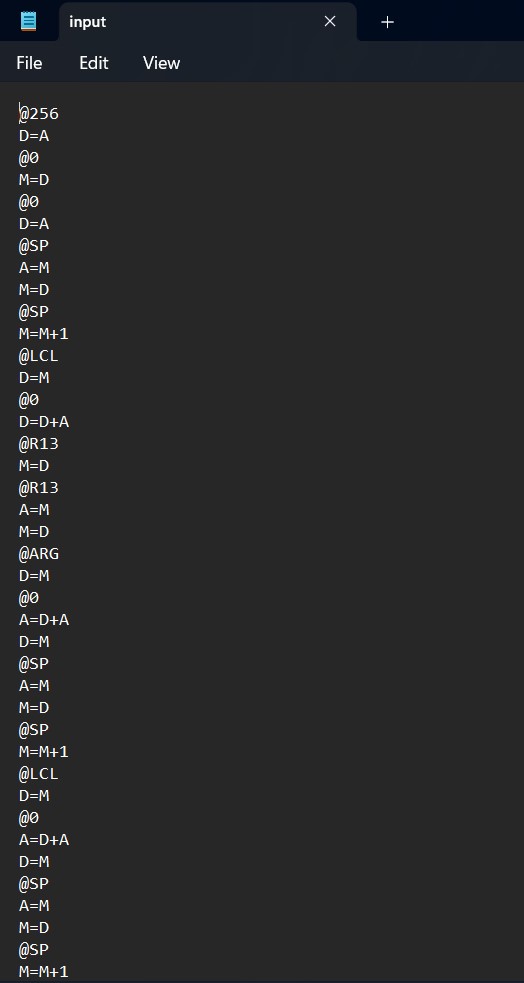
# RESULT

Converting BasicLoop.vm to BasicLoop.asm:

***INPUT:***

 *Figure 4. Input VM code*

***OUTPUT:***



*Figure*

*5*

*. Output ASM code*

# References

[1] N. N. a. S. Schocken, "The Elements of Computing Systems, Building a Modern Computer".