Name: MIDN 2/C Eshleman

Course / Section: SY303 / 3321

Enclosures: (1) Python Implementation of Assembler.py

(2) Screenshot for Assembler comparison of Add.hack

(3) Screenshot for Assembler comparison of MaxL.hack

(4) Screenshot for Assembler comparison of RectL.hack

(5) Screenshot for Assembler comparison of PongL.hack

**RUBRIC**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Section** | **0%** | **50%** | **80%** | **100%** | **Max Points** |
| Used Template | No | Partially |  | Yes | 10 |
| Introduction | Purpose and objectives unclear | Discussion lacking | Progressing | Well discussed | 5 |
| Procedure | Discussed procedure could not be reproduced | Some steps and tools missing | Progressing | Comprehensive and clear | 10 |
| Results | Not present or not discussed | Incorrect results without explanation | Progressing | Correct, well-explained results | 10 |
| Discussion | Questions not answered or answers are off-topic. | Discussions lack complete consideration. Terse responses. | Progressing | Well-developed discussions with interesting insights | 30 |
| Python Code | Missing or not commented | Incomplete, has errors, missing name or comments | Progressing | Includes identification and clear descriptions of all code | 20 |
| Assembler Screenshots | Missing | Does not capture full simulator screen or wrong | Progressing | All present, shows status of test script, full legible screen capture | 10 |
| Grammar/Professionalism | Poor grammar or use of slang |  | Progressing | Professional writing | 5 |

**INTRODUCTION:**

**The main objective of project 6 is to develop an Assembler that can translate programs that are written in Hack assembly language into binary code understood by the hack hardware platform. We have already built a portion of the program in our Lab 6. The already completed objectives that I created in lab 6 was the assembler can process A commands. The next step is to process C commands and labels that we can turn into binary code.**

**PROCEDURE:**

**In this project we had to add to our already functioning Assembler that we made in Lab 6, since the A commands were already programed we worked on the C commands for this project. The additional step we made in the main function of our program was to pass through any lines that were comments on our .asm files. We did this by first splitting the instructions so we could see if there was “//” . Once we found a commented line we would continue and not include that line of instruction. The main function remained unchanged from lab 6 otherwise.**

**For our parsing function we already had it determine if the line of instruction was an A command if the line had a @ symbol. The next step was to determining the C Commands, we did this by first determining if the instruction had an = or ; . If the instruction had an equal sign we split the line of instruction at the equal sign making a destination and a comp portion. If the instruction had a semi-colon we know that there was a jump in that line. If there was a jump we would spilt at the comp and jump portion to get their respective lines of binary code.**

**For the code function we already have the A command completed, so for the project we focus on the C Commands. For the C Commands we first add 111 to the beginning of the binary code to show that it is a C Command. Then we see if there is an M in the instruction line and we add either a one or zero for the A bit of our assembler code. Once that is completed we created a comp dictionary that has all of the binary code for the comp bits. We reference the dictionary each time and add the respective code to our mcode variable. Then we see if there is any jump commands in the instruction line, we created another dictionary that has all of the jump commands in binary stored. We reference that dictionary, and fill in the appropriate binary into our mcode variable.**

**After running through all of the different portions of the machine code our program is completed. Our newly created assembler can take a .asm file that does not have labels and can convert the file into a .hack with the appropriate machine code.**

**RESULTS:**

|  |  |
| --- | --- |
| Succeeded | Failed |
| * Add.asm * MaxL.asm * RectL.asm * PongL.asm |  |

If you were successful, describe an issue or a confusing concept you had to overcome to get this to work.

Before reference Dr. Brown’s video I was struggling with the code function of the program, I created the appropriate dictionaries but I was struggling with trying to actually create the machine code and add onto it. Once watching the explanation I was over complicating my code trying to search through the dictionary at each line with a lot of for loops, this made a lot of errors and would have taken a long time to process.

**DISCUSSION (For these questions you may seek quality information from the internet or other resources.  Be sure to cite all sources used for research, give the exact URL(s) visited):**

1. **Describe what an Assembler program is and how it works. How is it similar to and different from a Compiler program? Does one program contain a subset of the functions of the other, or do they contain disjoint sets of functionalities? The complier can check for errors in someone’s code rather than with an assembler it only translates code into the assembly language. Both an Assembler and a complier can take a high-level programming language and convert it into a machine level language. The complier has more subset of functions, it involves three different analyzers and also generates code, while the Assembler has two phases over the given input.**

**SOURCE USED: https://www.geeksforgeeks.org/difference-between-compiler-and-assembler**

1. **Section 6.3.5 in your text discusses the strategy for modifying your Assembler to handle Assembly code that includes symbols (which you can implement for Engagement credit!). The strategy recommends making 2 separate passes at processing the lines of code in your input file. Why is this necessary? Discuss the specific difficulty that would arise trying to assemble the Sum.asm program from your pre-lab if you tried to do so in one linear pass. With having the assembler handle symbols having two separate passes is necessary because symbols in the asm file are not defined until a later point and would give the wrong results. The assembler will use two passes through the first is with the assembler processes the inputs and then discards them expect for the symbols, then is goes through anther pass and inputs the symbol table that is already loaded.**

**SOURCE USED: https://homepage.divms.uiowa.edu/~jones/syssoft/notes/04fwd.html**

1. **Executable files on your computer can be examined in binary or symbolic form just as we have done with Hack code throughout this chapter. Install the analysis tools in your VM by opening the terminal and executing “**sudo apt-get install binutils**”. Now let’s examine the** ls **program on your system which lists files. Type the following command “**objdump -d -Mintel /bin/ls | less**”. This will open a view of the executable code which you can scroll through using your arrow keys (‘q’ to quit). Now search for the disassembly of the actual code in the text section, type “**/text:**” and hit Enter. Notice the 3 columns of information: instruction address on the left, binary machine code (in hexadecimal format) in the center, and the symbolic assembly instruction to the right. Take a look at the first several instructions (mostly stack** ‘push’ **instructions with different registers). What do you notice about the hexadecimal form of these** ‘push’ **instructions? Is there any order to the differences? Why can some** ‘push’ **commands be represented in 1-byte hexadecimal instructions while others take 2-bytes? NOTE: We will cover push commands in the next chapter, you don’t need to know *how* they work to look for patterns and think through this question.**

**I notice that hexadecimal form of the different push instructions go up by one each time. The order of the difference is the second digit in the hexadecimal format increase by one each time. I believe that while some push commands are represented by 1-bit hexadecimal instructions while others are 2-bits is because some instructions are bigger than the others.**

**COMMENTS:**

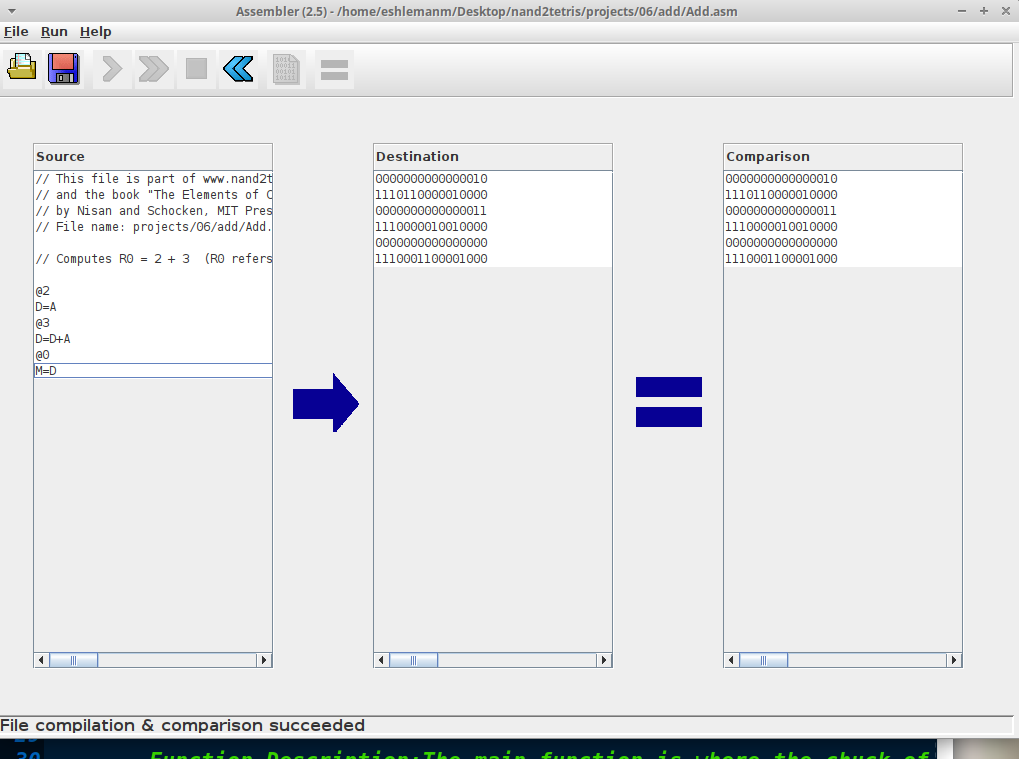
What did YOU gain from this project? If you felt like the project did not add to your understanding leave the feedback below. Your help is critical to improving the course!

I enjoyed this project because we coded in python, especially since I already had an understanding of this type of programming. This class is difficult because I am also taking SY301, so in previous projects I will write python code on accident because I am in that mindset with my other class. This project helped because I was not getting as confused. I am impressed with myself for getting this project to function properly.

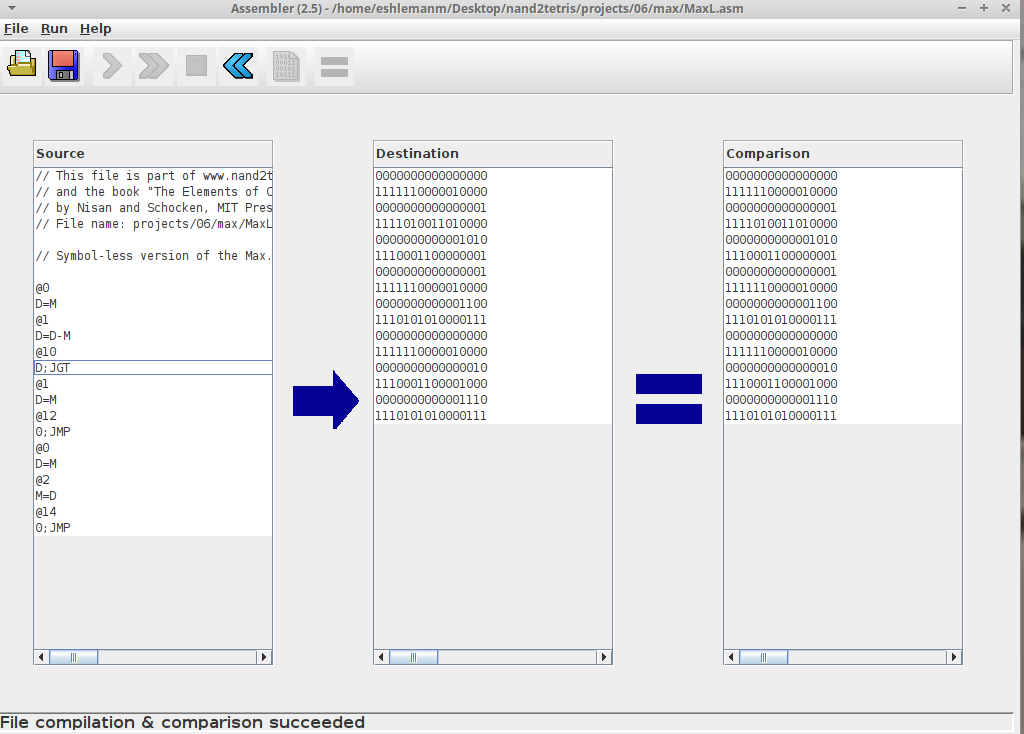
**ENCLOSURE (1): Assembler.py**

|  |
| --- |
| **#!/usr/bin/env python3**  **'''**  **Name: MIDN McKenzie Eshleman**  **Section: 3321**  **referenced Dr. Browns video**  **Description of Program:**  **'''**  **######################################################################################################**  **# Import any required libraries here**  **######################################################################################################**  **import sys #access command line arguments**  **dest ={"null":"000", "M":"001", "D":"010", "MD":"011", "A":"100",**  **"AM":"101", "AD":"110","AMD":"111"}**  **#Table of symbols that are used in assembly code:**  **symbols = {"SP": 0, "LCL": 1, "ARG": 2, "THIS": 3, "THAT": 4,**  **"SCRREN":16384, "KBD":24576}**  **def main():**  **'''**  **Function**  **Name: main**  **Function Description:The main function is where the chuck of the**  **program lies here we read and open the filename from the command lines**  **we also create a new file (f2) which will be the .hack file**  **Where the hack assembly code will be stored at.**  **'''**  **# Read filename from command line argument**  **filename = sys.argv[1] #./Assembler.py input.asm filename ='input.asm'**  **# Open the input .asm file with that filename**  **f1= open(filename, 'r') #creates a file and opens and reads the file**  **# Create and open the output .hack file with the same basename**  **f2 = open(filename.replace('.asm', '.hack'), 'w')**    **# Read each assembly instruction in the input file**  **# For each instruction:**  **for instruction in f1.readlines():**    **# For each instruction:**  **instruction = instruction.strip()**  **if not instruction:**  **continue**  **if "//" in instruction:**  **if instruction.startswith("//"):**  **continue**  **else:**  **instruction = instruction.split("//")[0].strip()**  **# Parse the instruction into its type (A or C) and fields and return them in a tuple**  **instruction\_type, instruction\_fields = Parser(instruction)**  **# Generate the corresponding 16 bit binary code of that parsed instruction as a string**  **machine\_code = Code(instruction\_type, instruction\_fields)**  **# Write that binary string to the output file**  **f2.write(machine\_code + '\n') #machine code \n after code function**  **# Be nice and close the files when you are done!**  **f1.close()**  **f2.close()**  **def Parser(inst):**  **'''**  **Function**  **Name: Parser**  **Inputs: inst**  **Outputs: Tuple, value**  **Function Description:**  **The main function of the parser is to break each**  **assembly command into its underlying components**  **(fields and symbols).**  **'''**  **# Your code here**  **inst = inst.strip() #Removes whitespace**  **if inst[0] == "@": #checks to see if inst has a @, THIS IS AN A COMMAND**  **return tuple(["A\_Command", inst[1:]]) #returns the tuple**  **else: # C instruction**  **if '=' in inst and ';' in inst:**  **dest, remaining = inst.split("=")**  **comp, jump = remaining.split(";") #spliting code if there is an = or ;**  **return ('C\_Command', [field.strip() for field in (dest, comp, jump)])**  **elif '=' in inst:**  **dest, comp = inst.split("=")**  **jump = "null"**  **return ('C\_Command', [field.strip() for field in (dest, comp, jump)])**  **elif ';' in inst:**  **dest = "null"**  **comp, jump = inst.split(";")**  **return('C\_Command', [field.strip() for field in (dest, comp, jump)])**  **else:**  **dest = jump = "null"**  **comp = inst**  **return( 'C\_Command', [field.strip() for field in (dest, comp, jump)])**    **#return # Return a tuple containing the instruction type and the separated fields**  **def Code(instType, instFields):**  **'''**  **Function**  **Name: Code**  **Inputs: instType, instFields**  **Outputs:**  **Function Description: The code function takes the hack assembly**  **code and translates it into 16-bit binary code.**  **'''**  **# Your code here**  **if instType == 'A\_Command':**  **binary = bin(int(instFields)) #converts to binary**  **binary = binary[2:] #removes the first two bits which are (0B)**  **binary = binary.zfill(16) #makes the binary code 16 bits long**  **return(binary)**  **elif (instType == 'C\_Command'):**  **mcode = "111" #adds 111 to the beginning of the machine code since it is a C instruction**  **dest, comp, jump = instFields**  **if "M" in comp: #adds a 1 for the A instruction if the C instruction has an M**  **mcode += "1"**  **else:**  **mcode += "0"**  **mcode += compDict[comp] #looks at the comp dictionary to add the correct binary values**  **if "A" in dest: #adds a 1 in the machine code if there is an A**  **mcode += "1"**  **else:**  **mcode += "0"**  **if "D" in dest: #adds a 1 in the machine code if there is an D**  **mcode += "1"**  **else:**  **mcode += "0"**  **if "M" in dest: #adds a 1 in the machine code if there is a M**  **mcode += "1"**  **else:**  **mcode += "0"**  **mcode+= jumpDict[jump] #refernces the Jump command dictionary to add the jump bits to machine code**  **return mcode**  **# Return the 16 bit string representing the machine code translation of the instruction**  **######################################################################################################**  **# Define any additional helper functions here (include fuction descriptions)**  **######################################################################################################**  **#dictionaries to translate the comp and jump portions of C-Commands**  **compDict = {"0": "101010", "1":"111111", "-1":"111010", "D":"001100",**  **"A":"110000", "!D": "001101", "!A":"110001", "-D":"001111",**  **"-A":"110011", "D+1": "011111", "A+1": "110111", "D-1":"001110",**  **"A-1": "110010", "D+A":"000010", "D-A":"010011", "A-D":"000111",**  **"D&A":"000000", "D|A":"010101", "M":"110000", "!M":"110001",**  **"-M":"110011", "M+1":"110111", "M-1":"110010","D+M":"000010",**  **"D-M":"010011", "M-D":"000111", "D&M":"000000", "D|M":"010101"}**  **jumpDict ={"null":"000", "JGT":"001", "JEQ":"010", "JGE":"011", "JLT":"100",**  **"JNE":"101", "JLE":"110", "JMP":"111"}**  **# Include the code below to automatically execute the main function when the program is run.**  **if \_\_name\_\_ == '\_\_main\_\_':**  **main()** |

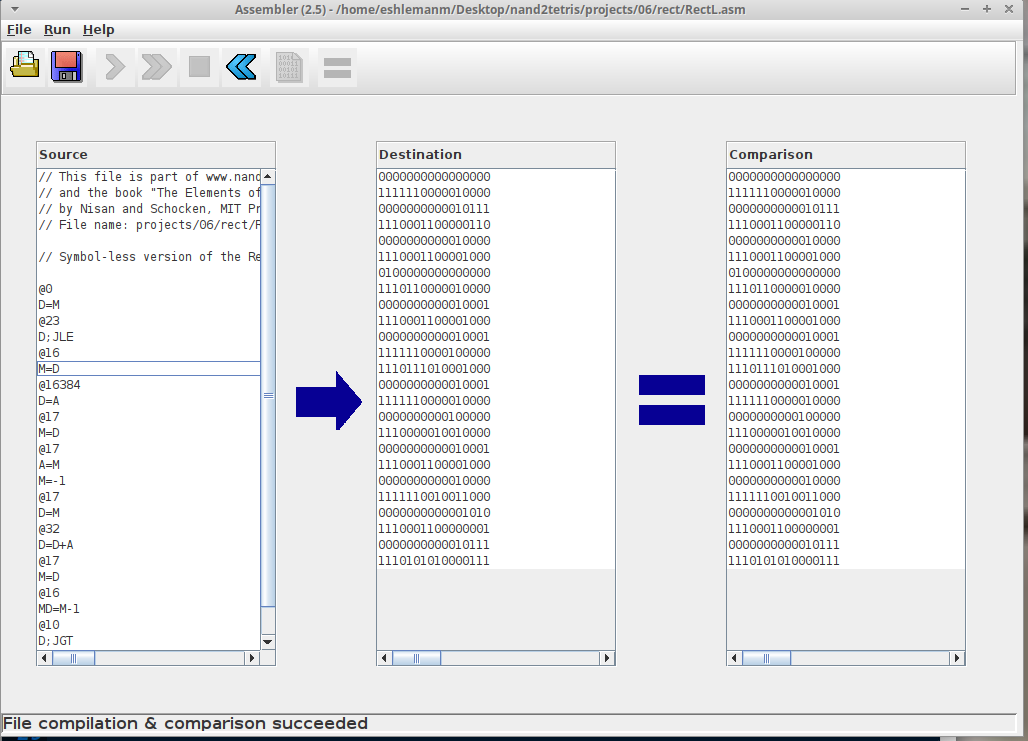
**ENCLOSURE (2): Assembler comparison of Add.hack**

****

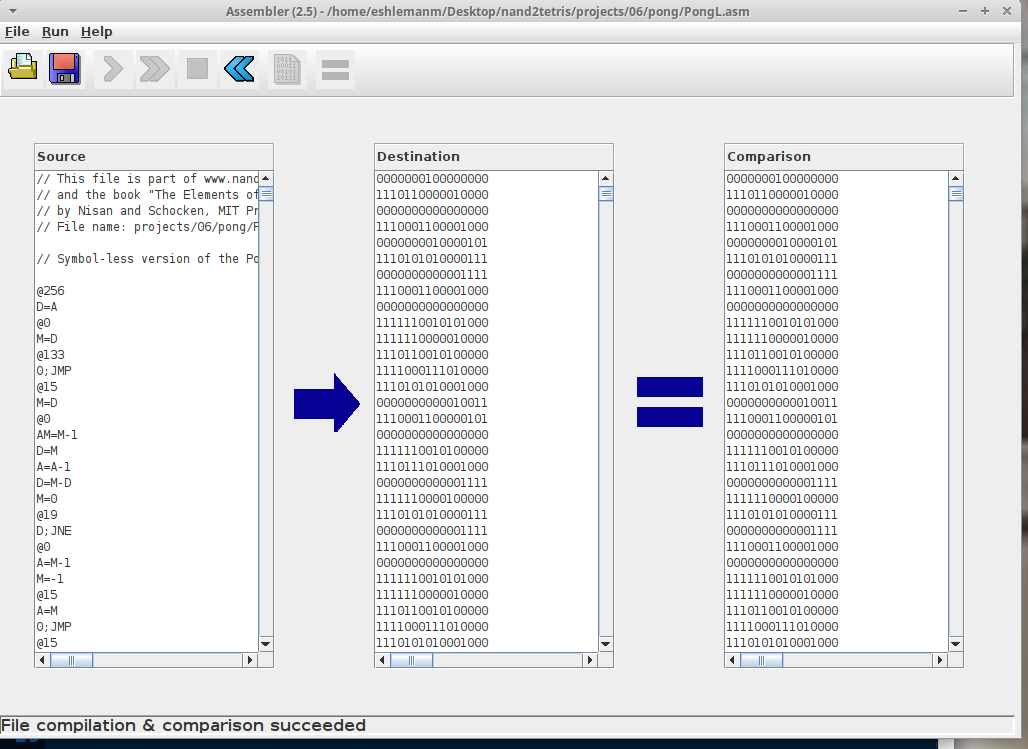
**ENCLOSURE (3): Assembler comparison of MaxL.hack**

****

**ENCLOSURE (4): Assembler comparison of RectL.hack**

****

**ENCLOSURE (5): Assembler comparison of PongL.hack**

****