Name: McKenzie Eshleman

Partner(s): Jon Robinson (out of class)

Section: 3321

**Objective:** In this lab, you will build a deeper understanding of how human written code is converted to machine executable instructions. You will learn how to translate Hack Assembly code into the bits that constitute each individual A-instruction or C-instruction, then write a Python script to automate this process for an entire Hack Assembly program. To receive full credit, properly commented code will be required (Python comments begin with “#”).

1. **Pre-Lab: Sum** 
   1. Begin with the **Sum** assembly program on page 65 that your instructor demonstrated for you previously. Copy this entire program into the table below and move all identified comments into the Comment column. Then identify the line number that will be associated with each line of code which **will** appear in **ROM** for execution, also identify the type of each command. The first few lines are filled out to help get you started. (10 pts)

|  |  |  |  |
| --- | --- | --- | --- |
| **Line #** | **Assembly Command** | **Comment (if present)** | **Command Type** |
|  |  | // Adds 1 + … + 100 |  |
| 0 | @i |  | A COMMAND |
| 1 | M=1 | // i=1 | C COMMAND |
| 2 | @sum |  | A COMMAND |
| 3 | M=0 | // sum=0 | C COMMAND |
|  | (LOOP) |  | L COMMAND |
| 4 | @i |  | A COMMAND |
| 5 | D=M | // D=i | C COMMAND |
| 6 | @100 |  | A COMMAND |
| 7 | D=D-A | //D=i-100 | C COMMAND |
| 8 | @END |  | A COMMAND |
| 9 | D;JGT | //If (i-100)>0 goto End | C COMMAND |
| 10 | @i |  | A COMMAND |
| 11 | D=M | //D=i | C COMMAND |
| 12 | @sum |  | A COMMAND |
| 13 | M=D+M | //sum=sum+i | C COMMAND |
| 14 | @i |  | A COMMAND |
| 15 | M=M+1 | //i=i+1 | C COMMAND |
| 16 | @LOOP |  | A COMMAND |
| 17 | 0;JMP | //Goto LOOP | C COMMAND |
|  | (END) |  | L COMMAND |
| 18 | @END |  | A COMMAND |
| 19 | 0;JMP | //Infinite loop | C COMMAND |

* 1. There is now enough info to resolve the label and variable symbols to their respective addresses in **RAM** and **ROM**. Given the pre-defined symbols, complete the **Symbol Table** with the user-defined symbols that would be generated based on this input program. Then, highlight all symbols that point to an address in RAM yellow and highlight all symbols that point to an address in **ROM** green. (10 pts)

Used Page 110 as a reference, there is a table stating this is all from RAM

|  |  |
| --- | --- |
| **Symbol** | **Address** |
| SP | 0 |
| LCL | 1 |
| ARG | 2 |
| THIS | 3 |
| THAT | 4 |
| R0 | 0 |
| R1 | 1 |
| R2 | 2 |
| R3 | 3 |
| R4 | 4 |
| R5 | 5 |
| R6 | 6 |
| R7 | 7 |
| R8 | 8 |
| R9 | 9 |
| R10 | 10 |
| R11 | 11 |
| R12 | 12 |
| R13 | 13 |
| R14 | 14 |
| R15 | 15 |
| SCREEN | 16384 |
| KBD | 24576 |
|  |  |
|  |  |
|  |  |
|  |  |

* 1. Complete the rest of the table below to decode each A Command and C Command identified in **step a** above. For each A Command, input its Instruction Prefix (0), the symbol if one was used, then the decimal and binary (15-bit) versions of the corresponding number. For each C Command, input its Instruction Prefix (111), then the fully decoded bits (a, comp, dest, jump) that make up the remainder of the 16-bit instruction. (10 pts)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Line #** | **Assembly Command** | **Instruction Prefix** | **A (symbol, if used)**  **C (a bit)** | **A (value–decimal)**  **C (comp bits)** | **A (value-binary)**  **C (dest bits)** | **C (jump bits)** |
| 0 | @i | 0 | i | 16 | 000000000010000 | 000 |
| 1 | M=1 | 111 | 0 | 111111 | 001 | 000 |
| 2 | @sum | 0 | sum | 17 | 0000000000010001 | 000 |
| 3 | M=0 | 111 | 0 | 101010 | 001 | 000 |
| 4 | @i | 0 | i | 16 | 000000000010000 | 000 |
| 5 | D=M | 111 | 1 | 110000 | 010 | 000 |
| 6 | @100 | 0 | 100 | 100 | 0000000001100100 | 000 |
| 7 | D=D-A | 111 | 0 | 010011 | 010 | 000 |
| 8 | @END | 0 | END | 18 | 0000000000010010 | 000 |
| 9 | D;JGT | 111 | 0 | 001100 | 000 | 001 |
| 10 | @i | 0 | I | 16 | 000000000010000 | 000 |
| 11 | D=M | 111 | 1 | 110000 | 010 | 000 |
| 12 | @sum | 0 | Sum | 17 | 0000000000010001 | 000 |
| 13 | M=D+M | 111 | 1 | 000010 | 001 | 000 |
| 14 | @i | 0 | i | 16 | 000000000010000 | 000 |
| 15 | M=M+1 | 111 | 1 | 110111 | 001 | 000 |
| 16 | @LOOP | 0 | LOOP | 4 | 0000000000000100 | 000 |
| 17 | 0;JMP | 111 | 0 | 101010 | 000 | 111 |
| 18 | @END | 0 | END | 18 | 0000000000010010 | 000 |
| 19 | 0;JMP | 111 | 0 | 101010 | 000 | 111 |

* 1. Analyze the executable code. Do these steps IN ORDER!
     1. Reassemble the 16-bit binary instruction from each line in your table above onto its own line in the left side box below; these lines would make up an executable **Sum.hack** program that you could copy into a file and then run in your CPU Emulator.
     2. In the next column, recreate the literal Hack Assembly command that would have created this instruction on the left if you had **no additional information** about symbols. This is called the disassembly; any machine code command can be reversed into an associated assembly command.
     3. In the final column, repeat the Hack Assembly code found in column 2, however if it is an A-instruction, check the Symbol Table from **part b** and replace the number following the @ sign with a symbol from that table if one is found. (10 pts)

|  |  |  |
| --- | --- | --- |
| **Sum.hack** | **Disassembled Sum.asm** | **Approximated Sum.asm** |
| **0000000000010000**  **1110111111001000**  **0000000000010001**  **1110101010001000**  **0000000000010000**  **1111110000010000**  **0000000001100100**  **1110010011010000**  **0000000000010010**  **1110001100000001**  **0000000000010000**  **1111110000010000**  **0000000000010001**  **1111000010001000**  **0000000000010000**  **1111110111001000**  **0000000000000100**  **1110101010000111**  **0000000000010010**  **1110101010000111** | **@16**  **M=1**  **@17**  **M=0**  **@16**  **D=M**  **@100**  **D=D-A**  **@18**  **D;JGT**  **@16**  **D=M**  **@17**  **M=D+M**  **@16**  **M=M+1**  **@4**  **0;JMP**  **@18**  **0;JMP** | **@i**  **M=1**  **@sum**  **M=0**  **@i**  **D=M**  **@100**  **D=D-A**  **@END**  **D;JGT**  **@i**  **D=M**  **@sum**  **M=D+M**  **@i**  **M=M+1**  **@LOOP**  **0;JMP**  **@END**  **0;JMP** |

1. **Assembler: A-instructions**

**Purpose:** Describe the function and purpose of the Assembler in your own words. What are the inputs and outputs? (5 pts) The assembler is a text-processing program, essentially the assembler takes in an .asm file and then translates it into hack assembly code and then will create a new file with the assembly code in a .hack file.

**Setup: Save the Assembler.py template from Blackboard Project 6 assignment into your nand2tetris project06 folder. Also save the NoSymAonlyL6.asm file to the same location for testing later.**

**Open the Assembler.py file in your text editor (vim). Fill out your name, section, and description information.**

We will build the solution for assembling A-instructions logically, one step at a time.

**HINT:** This is a rudimentary solution just to understand how to process A-instructions. Your final project submission will need to be more robust and deal with more possibilities, such as inline comments.

**Step 1**) The first comment asks you to import any required libraries. You will need to read the name of the input file from a command line argument. Write the Python statement to import the library which is required in order to access command line arguments. (2 pts)

|  |
| --- |
| **Import sys** |

**Step 2**) Next we will work on the **main** function where you will setup the File I/O for the input assembly file and the output hack file. Then we will read the lines of instructions from the input file in a loop. Within that loop we will send each line for processing by the **Parser** function and the **Code** function, then write the generated 16-bit string to the output file.

**Step 2a**) Fill out the function description comment block for **main**. This needs to include your own words, not mine. (1 pts)

|  |
| --- |
| **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.** |

**Step 2b**) The name of the input Hack Assembly command will come from the only command line argument. It will NOT be HARDCODED! Write the Python code below to save this argument to a variable representing the input file name. (2 pts)

|  |
| --- |
| **Filename = sys.argv[1]** |

Now, since we will use the same base name for the file to create a .hack file for output, write Python code that will save the base name to a string that excludes the “.**asm**” extension.

|  |
| --- |
| **F2 = open((filename+ “.hack”), ‘w’)** |

**Step 2c**) Write Python code that will open the input .**asm** file in read mode with the filename you saved earlier. Save this input file object to its own variable. (2 pts)

|  |
| --- |
| **F1 = open((filename, ‘r’))** |

Now write Python code to open the output .hack file in write mode. The file name must include the base name you saved moments ago concatenated with “.**hack**”. Save this output file object to its own variable.

|  |
| --- |
| **F2.write(machine code)** |

**Step 2d**) You must now implement the for loop to iterate through each line of Hack Assembly code in the input file, convert it to Hack Machine Code, and write the resulting 16-bit string to the output file. Write a Python **for** statement that will iterate through the lines of your input file, storing the contents of each line in a variable named **instruction**. (4 pts)

|  |
| --- |
| **For instruction in f1.readlines()** |

Now write the internal statements of this for loop which are already included in the template. Add code after this which expects the **machine**\_**code** variable to contain the 16-bit machine code string for the current instruction (ie. “**0110110110110110**”) and writes this result as a line to the output file. **REMEMBER**: **for** loop instructions are expected to be indented consistently!

|  |
| --- |
| **instruction\_type, instruction\_fields = Parser(instruction)**  **machine\_code = Code(instruction\_type, instruction\_fields)**  **f2.write(instruction)** |
|  |

**Step 2e**) At this point, the main program is finished and ready to exit. As a good programming habit, include Python code to close the files you have been working with. (1 pts)

|  |
| --- |
| **F1.close()**  **F2.close()** |

**Step 3**) Now move on to the **Parser** function to process a single line of Hack Assembly and return the type of command it is as well as the parsed fields which will need to be decoded by the **Code** function.

**Step 3a**) Fill out the function description comment block for **Parser**. This needs to include your own words, not mine. (1 pts)

|  |
| --- |
| **The main function of the parser is to break each assembly command into its underlying components (504050**  **fields and symbols).** |

**Step 3b**) Now it’s time to fill in the **Parser** implementation code for an A-instruction. Your argument string is named **inst** (for instruction). It contains a line of code from the input file which may have excess white space before or after it. Write a line of Python code to remove any extraneous white space. (1 pts)

|  |
| --- |
| **Inst = inst.strip() #Removes whitespace** |

**Step 3c**) What element will always be present in a Hack Assembly A-instruction? (3 pts)

|  |
| --- |
| **If the hack assembly has an @ then it is an A instruction, if not it is a C instruction** |

Write a Python **if** block which will check if **inst** is an A-instruction by verifying the presence of this element. If it is an A-instruction, return a 2-tuple containing the string “A\_**Command**” as well as the string representing the value portion of the A-instruction.

|  |
| --- |
| **If inst[0] == ‘@’:**  **Return tuple([“A\_Command”, inst[1:])** |
|  |

**Step 4**) Now move on to the **Code** function to process the field(s) of a certain instruction, given its type.

**Step 4a**) Fill out the function description comment block for **Code**. This needs to include your own words, not mine. (1 pts)

|  |
| --- |
| **The Code function takes the hack assembly code and translates it into 16 bit binary code.** |

**Step 4b**) Write a Python **if** block that will check if the instruction type is an A Command. If so, there should only be one field. Convert the value in that field to a binary string and make sure it is 16 bits long, fill the most significant bits with ‘**0’** as necessary. Return this 16-bit binary string. (4 pts)

|  |
| --- |
| **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)** |

**Step 5**) You have now defined 3 functions, but have not told Python what you want to actually execute. At the very bottom of the template is a comment that asks you to include a Python **if** block to run the main function when the program is executed from the command line. Include this code here (this is common Python code that you have included in many previous assignments). (3 pts)

|  |
| --- |
| **If \_\_name\_\_\_ == ‘\_\_main\_\_’:**  **Main()** |

1. **Test Your Assembler**

Save your Python Assembler.py script and exit the text editor. Make your script executable if you had not done that yet. Now run the following command in the terminal in the project06 directory where you saved the files from earlier**.**

|  |
| --- |
| **./Assembler.py NoSymAonlyL6.asm** |

Resolve any errors generated while executing your code.

Once the errors are resolved and your program runs successfully, run the “**ls**” command and verify that the “**NoSymAonlyL6.hack**” file was properly created.

Now open the Assembler from the nand2tetris **tools** directory.

* Choose “**Load Source File**” and choose **NoSymAonlyL6.asm**.
* Choose “**Load Comparison File**” or hit the “**=**” icon and choose **NoSymAonlyL6.hack**.
* Step all the way through the comparison until you see the message “**File compilation & comparison succeeded**”
* Paste a screenshot of the whole Assembler window showing this message. (10 pts)

|  |
| --- |
|  |

1. **At the end of the Pre-Lab, you disassembled the Sum.hack executable file and attempted to associate addresses with symbols in column 3. How close was your column 3 code to the original Sum.asm? If it was perfect, is that only because you looked back at the code? Discuss the error inherent in this disassembling process given ZERO knowledge of the original code. What could go wrong? What if you were asked to disassemble a program with NO Symbol Table?** (10 pts)

**My column three code was the same as the previous charts that we created. It was helpful to look back at the code, however a pretty simple mistake that could occur is trying to include the labels like (LOOP) and (END). If I did not have any knowledge of the program I would make the mistake of trying to include the labels into the disassembled program.**

1. **Understanding how a compiler or assembler translates code lets us optimize code when we write it. What other benefits are there to understanding how an executable file was compiled or assembled? (HINT: Think like a security researcher)** (10 pts)

**The complier can check for errors in someone’s code rather than with an assembler it only translates code into the assembly language. If I was to try to hack someone I would want them to only assemble their code because then I could find the errors and try to exploit them to my benefit. Over all a compiler is more ideal because it makes sure there are not any errors.**