Author 1: Altay Acar - 2018400084
Author 2: Engin Oğuzhan Şenol - 2020400324

CMPE 230

Systems Programming

Project 2

Introduction

In this project, we are asked to convert an assembly source code to an output. As an input we get an `asm` file that contains lines of assembly code. In order to reach the goal first, we needed to build an assembler to produce 6-digit hexadecimal numbers that translates into 24-bit binary number unique to each input line. Then, we needed to create an execution simulator which can execute these hexadecimal numbers to produce an output.

Assembler

First, we created a function `is_hex(s)` to check a given string is hexadecimal or not. Every register and instruction have their unique hex code. To create their corresponding 24-digit binary encoding we needed to get their hex codes as we encounter them in input. We stored this data in a dictionary named `data`. When registers are given as key 4-digit hexadecimal number is returned and when instructions given as key 2-digit hexadecimal number is returned.

We processed the input and removed every empty line and space if any is encountered and then tokenized every line. `commands` list is a list of lists that contains every line in tokenized form. `Labels` dictionary stores every label and the memory address that the label points to as key and value pairs. Labels cannot be declared more than once. So, if we encounter a duplicate label, the program produces no output and prints an error message. We created a wide for-statement which checks every line of input and gives the corresponding hexadecimal value as output. First, we create variables called `opcode, addrmode, and operand` and set them to the value of -1 as their initial values.

Since every instruction, besides `HALT and NOP` needs another variable as operands with them, we put an if-statement that checks if the length of the line is two, one, or not. For the lines which have two tokens inside it, we check if their second token represents a register, memory address, immediate data, ASCII character or a memory address stored in a register. After detecting the type of the operand, we assign its corresponding hex codes in `addrmode and operand`. Since the first word is instruction, we use `data` to find its corresponding value and we give that value to `opcode`.

For `HALT and NOP`, we implemented separate but similar elif-statement because they are in a one token sized input lines. Before the wide for-statement reaches its end, we combine the values of opcode, addrmode, and operand to produce corresponding 24-digit binary output and converts it into 6-digit hexadecimal number. Our code repeated this statement for every line until it reaches to the end of the lines. The last output, which is 6-digit hexadecimal values, will be the input of our execution code.

Author 1: Altay Acar - 2018400084
Author 2: Engin Oğuzhan Şenol - 2020400324

Execution

In this execution simulation of CPU and memory structure of a computer we approached the registers in CPU230 and memory as data storage structures. So, before the execution part starts, we declare dictionary structures `registers` with register name and the hexadecimal number it stores as key-value pairs, `find_reg` with register's hex encoding and the name of the register as key-value pairs, and `flags` as flag name and its binary state (0 or 1) as key-value pairs. Initially we set the data that registers store flags to zero. While PC points to the start of the memory, stack pointer S points to the end of the memory, which is 0xfffe. Then we create a memory list with the size of 65536 (for 64K) and fill every memory cell with the value of zero in 8-digit binary number form. This helped us not to have any trouble when we try to reach any cell in our memory.

After that, we create a for-statement which fills the necessary memory cells with the values of the inputs. Since every memory cell has 8-bit size and, we have 24-bit inputs, we divide every input to 3 parts that are 8-bits wide and place them into memory cells starting from the zero in order. This placing is applied for every instruction line. Also, we wrote another if-statement which checks if the length of the memory is reached. This statement gives an error message and exits from the execution if the memory limit is exceeded.

If an input did not contain the HALT function, it would be a problem for the code to run. It can enter an infinite loop or never exits to the system. To check this condition, we created a variable named `found_halt` and set it to 0. Then we created an if-statement which can detect if the input has halt or not. This statement searches the 24-bit binary version of 6-digit hex numbers. Our variable is set to 1 if it finds the value corresponding to the HALT instruction, which is 000001000000000000000000. If not, it gives an error message and exits execution.

Since we cannot fill a memory cell with 16-bit binary numbers, we create a `div_two` function which converts the 16-bit value into a list of two 8-bit values. When storing a 16-bit binary number in memory we use this list of two 8-bit values and store them in nth and (n+1)th memory cell. Also, when we need the value of operand (16-bit) that is stored in nth memory cell, we need to combine the values that are stored in nth and (n+1)th memory cells. For this operation, we used a `take_val_from_mem` function.

Since we have to use them in most of the instructions, we created two functions, `add_opr` and `not_opr`. While `add_opr` function takes two integers and returns to their sum, `not_opr` function takes an integer and returns to its complement. Flags are set accordingly for both functions.

Our other two function is called `shl_opr` and `shr_opr`. These functions first take a hex value and change it to a binary value by shifting that value by one position to left and right respectively. Flags are set accordingly for both functions.

Then, we write a variable called "execute" and set it to the value of 1. We implement a while-loop that is executed as long as execute is 1 i.e., until reaches a break statement. In this while-loop we performed every instruction's operation ordered in if-statements.

Author 1: Altay Acar - 2018400084
Author 2: Engin Oğuzhan Şenol - 2020400324

Problems We Encountered While Implementing

While we were implementing the project code, we face with several problems that we have found a solution at the end. Some of the problems were listed below, with their corresponding solution name:

`cmp_above`: Before JA, JAE, JB and JBE instructions' implementation, there must be a CMP instruction to set the flags. For this, we create a variable called `cmp_above` and set it to the value of zero. This variable. This variable is set to 1 when CMP instruction is executed and is set to 0 when another instruction is executed. If we do not change it to zero again after at least one instruction is executed after CMP, it would stay 1 until it reaches a conditional jump and perform the jump even though CMP instruction is not right above the conditional jump statement. To adjust this, we wrote `cmp_above = 0` after every probability of every instruction besides CMP. Hence, we were able to detect if the previous instruction was CMP or not.

Inputs with empty lines: When we were trying the testcases we encountered an error with the inputs that includes empty line(s). At first these empty lines acted as they occupy memory cells because labels' addresses that they point to are calculated with the amount of lines above them given in input asm file. Then, we implement

```
lines = list(filter(None, lines))
```

code which deletes the empty lines from input. Thus, we prevented a possible error by changing the input format while processing it.

Parts of the Code That Can Be Improved

Although our code works just fine with given inputs and some other testcases that we tested, the amount of syntax error types can always be infinite. We tried to check as much syntax errors as we can, but we think some erroneous input can still be given. If we had more time to spend on this project we could think of some more syntax error probabilities and handle them with care.