Umut Deniz Şener 2018400225
Leyla Yayladere 2018400216

CmpE 230 Systems Programming,

Spring 2021 - Project 2

Project Report

1.Problem Description:

Implementing an assembler and an execution simulator for hypothetical CPU called CPU230. Each instruction for this CPU is 3 bytes with encoding opcode in the first 6 bits, next 2 bits addressing mode and last 16 bits operand. Memory of CPU230 is 64K. Registers of CPU230 are called PC, A, B, C, D, E, S. PC is program counter and S is stack pointer. We have 28 different instructions which are requiring immediate, memory, register or combinations of these as operand. Also there can be labels in source code between instructions which should be defined the following format "LabelName:". First, we should translate the source code (*.asm) to binary file (*.bin) which encodes the instructions as hex number by **cpu230assemble**. Then, we should execute this binary file and produce output file (*.txt) by **cpu230exec**.

2. Problem Solution:

Firstly, we traversed the source file in order to find the labels and to assign memory address to them. Then, over again we traversed the source file line by line and looked the type of instructions and their operands and the type of addressing mode by tokenizing the line. We store the code for different instructions and registers in dictionaries in order to make our code simple. We determined 3 different hex number for each line which correspond to instruction code, addressing mode and operand. Then, put these hex numbers to converter function and get 1 hex number for each line and write this number to binary file(*.bin). We detected errors (e.g. 2 different assigning for one label, trying to jump undefined label or undefined instructions format etc.) if there is any in source code with if-else statements.

Second part of our project is execution. Firstly, we take hex number from the binary file for each line and convert it to binary number and separate these 24bits binary number into 6bits opcode, 2bits addressing mode and 16bits operand. Then, we find the corresponding instructions for these binary numbers. We store the values of registers and flags in dictionaries and setting these values according to

our binary file. We implement the memory stack through an array called **stack** and size of the array is 65536 (65536byte = 64KB). First, we put the instructions into this array from its beginning. In our stack, we store the data as binary number and we do all calculations in binary number format. We write 3 different function in order to apply binary number calculations. One of them is called **compelement**, takes 16bits long binary number and return its 1's complement form. Other one is called **add**, it takes 2 binary numbers and return their sum in binary format. Last one is called **carry**, it takes 2 binary numbers and return true if their sum in binary format has carry, false if it has not carry. We implement subtraction in the form A – B = A + not(B) + 1. We easily implement the instruction via if-else statements thanks to doing all calculations in binary and our functions. We do jump statements according to flag information in our dictionary. We set these flags after instructions properly. In push and pop statements, we point the end of our memory stack/array through S register and incrementing/decrementing this register value properly.

We implement the "READ" instruction such that it reads the first character from the console, takes input with pressing enter.

3. Conclusion:

- Firstly, we did calculations in different number format such as binary, hex, decimal but then we got confused. We switched every number to binary and do all calculations in binary format which is requested. In this way, everything becomes easier.
- Thanks to dictionaries and python language, it is quite readable.