Buse Tolunay 2018400195
Begüm Arslan 2018400189

Assembler and Execution Simulator Project

Our Project was to implement an assembler and an execution simulator for a hypothetical CPU called CPU230 whose rules are given by Prof Özturan.
So it has two parts:

1-CPU230ASSEMBLE

There were 28 different instructions to handle. We read the input line by line and decide the on the values of opcode, addressing mode and the operand. We used one dictionary to hold instructions and their opcodes, one for registers and their numerical values(as decimal), an done more for labels. We decided to make our program case insensitive. The problems were:

a- Labels:
We need to know where are the labels, so we keep their address values as decimal in a dictionary in the first traversal of the input file. When there is a jump to that label, we look at that map. Label names should not start with a number and also other syntax related to labels are checked in the first traversal.

At the second traversal we made the transition i.e. assembling.

b- Hex numbers:
Hex numbers are immediate values. We first check if the operand is a hex number with the checkHex() function. If it is we keep that number as a decimal value.

c- ‘ ’:
Another immediate date is the characters. Characters were given inside single quotes.

d- []:
This means there is an address inside the []. If the value here is in registers, addrmode is 2, else if it is a hex value addrmode is 3, else there is something wrong.

e- Tokenizing:
We simply used Regex as suggested

f- Lastly operand can be a label or a register. Than their value is taken from the related dictionary.

2-CPU230EXEC

To simulate the memory and registers of a computer we created a library called “memory” that is composed of 1-byte memory blocks and we did required modifications in the memory using it. We applied the same logic to “flags” library which contains flags as variables. Also we created registrations and instructions library to match them with their key numbers.

We read from the inputfile line by line.Then we converted every instruction into its binary form ,split it into three 1byte-parts and loaded them to consecutive memory addresses. (memory[i], memory[i+1], memory[i+2])

To execute the instructions we combined three consecutive memory addresses and again split them into three parts as operation code(6 bit), addressing mode(2 bit) and operand(16 bit). To realize the instructions we called “func” and gave opcode, addrmode and operand as parameter. After executing each instruction we incremented the pc(program counter) by 3 since a instruction is composed of 3 byte information.

Split: Very useful function to load information to memory.Takes a value as a parameter and converts it into binary form .Since the value contains 2 byte information it splits the binary form of the value into two parts and converts them into integers .
imr: Instructions that has valid immediate ,register and memory addressing modes calls imr . The function returns the operand depending on the addressing mode.

Mr:Returns operand for instructions that memory and register addressing modes are valid.The function return the memory address of the operand . Instructions with immediate addressing mode can not call this function.

Pair:Takes operand address as its only parameter and returns the value that is stored in the parameter and the following memory address(parameter+1).

Func: Parameters are opcode ,operand,addrmode of an instruction.It calls the functions or does the necessary changes in the memory according to its parameters.Creates if cases based on the instruction’s operation code and calls imr or mr depending on the addressing mode.

We used _Getch class to read a single character from standard input and cited the web address.
https://stackoverflow.com/questions/510357/how-to-read-a-single-character-from-the-user