1. Problem Description

In this assignment, we are asked to implement (i) an assembler and (ii) an execution simulator for a hypothetical CPU called CPU230.

For the first part of this assignment we are asked to write an assembler which assembles AT&T assembly language instructions to 3 byte long hexadecimal .bin outputs.

Every line consist of opcodes followed by an operand or just a label name. Combining these 6 bit opcodes, 2 bit addressing mode of operand and 16 bit operand itself led to 24 bit instruction total. These binary sequence converted to hex values before outputted .bin file.

After we run the assembler with the given .asm input and produce the corresponding .bin file, we move on to the second part of this project.

In the second part our aim is to carefully interpret the .bin file and make the necessary calculations carefully.

CPU230 instructions consists of 3 bytes. First six bits of the first byte corresponds to instruction opcode which helps us to distinguish what statements we need to execute for current instruction. Last two bits of the first byte corresponds to addressing mode. In each case of addresing modes, our approach to get the operand is different. Remaining two bytes of the instruction is called operand. Operand can be an immediate data, an encoding for a register, a memory address based on the addresing mode.

CPU230 has three flags: Zero flag, carry flag and sign flag, ZF, CF ,SF respectively. In instructions where an arithmetic operation is needed we set this flags accordingly so that when we reach a jump instruction our execution simulator works as intented.

There are seven registers associated with CPU230: PC, A, B, C, D, E ,S. Each of them has its own representation in binary. S refers to the stack pointer which points to the end of the memory.

2. Problem Solution

First part of the assignment is just a translation from assembly language to its corresponding binaries.

This translation starts with reading given .asm file to check and store the labels in the program. While reading line by line, if there is a line which includes a colon, then store this line without a colon as label to use it in the next part of the first part of the assignment.

After storing all labels to a map, read the input from scratch. While reading file line by line, append the translated output to .bin file after every line.

There are 28 instructions as given in the description. After deciding the instruction and its code (6 bits), determine the addressing mode (2 bits) of the operand which can be Immediate Type, Register Type, Memory Address Type, Memory Address in Register Type. After finding opcode and addressing mode, then find the 16 bit long corresponding of the operand. All these finding processes are done with regex.

When all 3 bytes are set, we are good to go. Process these bits and convert them into hexadecimal values. Then write these hex values to .bin file.

For the second part of the assignment, first of all we need to read the .bin file. While reading, since .bin file format is hex, we need to convert it from hex to binary. To make this job easier, we defined a dictionary object called *hexToBinary* which holds hex numbers as keys and their binary forms as pairs. When the convertion is done, we put the binary instructions to memory. Tricky part here is memory cells can hold 1 byte each while instructions are 3 bytes. Therefore we need 3 memory cells consecutively to store 1 instruction.

Since we are done with converting and putting instructions to memory now we need to begin the simulation. Each instruction has its own meaning and therefore requires different operations. However, there are 28 different instructions and explaining each one of them is ineffective because the procedure we follow is clear for any instruction.

Before explaining the general procedure, our memory cells are registers have the initial values binary 0 string except S which points to the end of memory.

What we do for each instruction is more or less same,firstly we need to decide its opcode by looking at first six bits. Operand and the actual operand in these instructions are different. Operand is given by the last two bytes of instruction. What we mean by actual operand depends also on addressing mode. For instance assume addresing mode is “10” then our actual operand is in a memory cell pointed to by given register as operand.

Opcode(first 6 bits) -> turn it into decimal and check with if statements

Addressing mode(last 2 bits) -> by using these modes get the actual operand for calculations

Operand (remaining 2 bytes) -> use it to get actual operand depending on addressing mode

Once we get the actual operand, its time to perform operations. These operations depends on the opcode but for each case what we do is to do necessary operations. If it is an arithmetic operation set the flags accordingly after calculation, if we are to store the result somewhere do it.

One topic needs to be discussed is how READ operation works. In our project READ expects 1 character input such as ‘r’, ‘R’, ’-’. Inputs which contain more than 1 characters are invalid according to our implementation. Please consider this while running testcases with READ operation

3. Conclusion

After trying the code with the given testcases and more additional testcases that we have written, we are of the sense that the assembler and execution simulator works well. The .bin files succesfully generates the wanted .txt files when simulated. Using regex in the first part and dictionaries in both parts decreased the number of lines written and increased the readibility of the program.

Ali Kaan Biber 2018400069

Yavuz Samet Topçuoğlu 2019400285