ERIM ERKIN DOGAN
2019400225

CMPE230 PROJECT-2

Project Description

In this project, we are asked to make an assembler and an execution simulator for a hypothetical CPU named CPU230. To achieve these, we need two different programs: one assembler named **cmpe230assemble** and one execution simulator named **cmpe230exec.**

CMPE230ASSEMBLE (ASSEMBLER)

Problem Description:

In this module for project, we need to parse and convert the given instructions to 6 digits hexadecimal instructions outputted in a bin file. There are some problems that needs solving such as finding labels' address and deciding operands' type.

Approach:

First, we need to determine memory values of Labels, to solve this we need to iterate over the input two times. First to determine memory address of Label and second to parse the input. The parsing has a caveat in which we need to determine the operands' type as: Immediate, Register or Memory. To solve this need to set some distinction between operands, which will help greatly to create binary instructions from given assembly code. So, we set some rules which will help us determine type of operands and parse the input:

1. Labels start with an alphabetic character and consists of alphanumeric characters.
2. Label declarations ends with ':' and there will be no other words/elements in that line.
3. There are 6 registers with hex encodings such as A=0001, B=0002, etc.
4. Memory type operands starts with '[' and ends with ']', can take hex or register inside the brackets
5. Input is not case sensitive except given immediate characters between '' symbols and they are stored with their ASCII codes.
6. Immediate hexadecimal operands start with a number
7. There is only one instruction per line

Implementation:

Starting our implementation of assembler; we first need to check if input file is of the true format, so we first check the extension of input file. Then if the input type is correct, we can start to pre-process the input file which will determine our Labels and their memory addresses. Pre-process part is just a full loop line by line iteration of the input file. We increase our memory index by 3 for every instruction and determine Label memory address by assigning the index. While doing this we also check validity of Label declaration as if it is declared twice or complies with rules given in approach part, if not the program gives an error and quits.

For the second part, we need to parse rest of the program but we have dozens of instructions with multiple supported operand types so using if-else for every instruction and operand type will make the code a bit messy. To solve this, we used dictionaries which groups up instructions according to their accepted operand types. So, using this method we check operands and their data types in if-else control flow chains instead of instructions resulting in much more readable, simple code.

Checking operands for type is straightforward using the assumptions we had earlier. Memory type operands have '[' and ']', register type operands have case insensitive names A, B, C, D, E which is converted to uppercase to achieve case insensitiveness. Immediate operands have 3 types: Labels, hexadecimal data and characters. To detect the type of operand, the program first checks if the operand have ' in it, then if the operand is of type character which will be converted to its ASCII code which will be encoded in binary; program checks if the operand is a label by checking if our predetermined labels from pre-processing. Determining hexadecimal immediate operands is a bit more tricky, it needs to start with a number and it needs to have only hexadecimal characters ([a-f][0-9]). After determining the type of operand, the program checks if the instruction supports use of the type by checking our dictionary and getting binary encoding of instruction. Then the binary encoding of 6bits instruction converted to hex after combining it with binary encodings of operand and operand type. Then the hex code is outputted to a "bin" file which in turn will be used to run our program with CMPE230EXEC.

Order of determining operand and creating binary instruction:

Instruction with no operands -> memory type operand -> register type operand -> immediate character type operand -> immediate label type operand -> immediate hex operand

ERIM ERKIN DOGAN
2019400225

CMPE230EXEC (EXECUTION SIMULATOR)

Problem Description:

For this module, we have a hypothetical CPU and 64K memory used to simulate given program, which is encoded in 6 hex numbers having 24bits length. To simulate this, we need to handle memory, registers, pointers, and stack of given CPU and interpret the input to run instructions.

Approach:

We firstly need to create elements of given hypothetical CPU, registers, flags and memory, in our code so that we can imitate their behavior using these elements. The input needs to be parsed and inserted to memory. Then we need to specially handle every instruction and operand type which results in an if-else chain, checking instructions and handling their expected behaviors. To achieve this using OOP approach, we need decide common parts of the instructions which is data storage and data retrieval. For these operations and instruction behaviors we need to assume some rules and errors:

1. Addresses which contain program instructions are read-only
2. Stack pointer's address can't be in program instruction zone or smaller than PC's current address
3. Can't assign data in stack's area other than POP and PUSH operations
4. Programs runs until it gets an illegal instruction, HALT instruction is called, or PC meets S

Implementation:

In implementation part we mostly followed the steps given in approach part. Firstly, we created 3 lists containing instructions encoded in 6bit binary categorized by operand types, like the dictionary solution in cmpe230assemble; then I set up memory consisting of 1byte blocks for a total size of 64K filled with default value of "00". Memory carries 2 characters hex code for one block, so instructions are divided into 3 blocks as they consist of 24bits/3bytes. Then we set flags as variables that will carry 0 or 1 representing true/false. Normal registers are stored in a dictionary with keys as their integer encoding, PC and S are set up as variables containing their data as integers. Also, there is a variable called program_size which keeps track of number of memory blocks occupied with instructions.

To start simulating, we first parse input file that carries hexadecimal 24bit instructions, then the program checks the size of instructions and assigns them to memory in order, while adjusting program_size accordingly.

After parsing the input and the program starts to iterate over memory starting from beginning of the memory adjusting PC and reading instructions from memory. Then the instructions are split into instruction, operand type and operand. Then the program checks the instruction to achieve wanted behavior. There are functions for general data load and store operations as instructions support multiple types of operands which employs created lists for categorizing instructions by their operand type. Also, there are specialized data storage and load functions for registers and memory which utilizes the rules given in approach.

ADD, SUB, NOT and JMP instructions have their generalized functions as their variations are used inside other instructions such as ADD is used in SUB, INC; SUB is used in CMP, DEC; NOT is used in SUB; JMP is used in other jump instructions with conditions.

PUSH and POP instructions stores and loads data from stack which is situated at the end of the memory, if a input tries to POP an empty stack “Stack underflow” error will be given, or if stack size get too bigger to meet PC or get out of memory bounds than “Stack overflow” error will be given.

READ instruction takes input till there is a character input other than new line. If the input is more than one character long, it stores the first character. Valid input examples:

>>C >> (whitespace) >>P READ: ‘C’ READ: ‘ ’ READ: ‘P’	>>C M P E 2 3 0 READ: ‘C’
	>>CMPE230 READ: ‘C’

PRINT instruction will print the operand as a character using ASCII conversion in an output .txt file with same name as program input file.
Program’s quit conditions:

1. HALT instruction
2. Error (Invalid input, Illegal Instruction)
3. End of memory
4. PC meets S (Stack Pointer)

Project Conclusion

For the conclusion, the project was straightforward in implementation side as we did the nearly same thing in Project 1, but it was also easier because of using a high-level language like Python which makes tokenizing and parsing input easier with built-in functions. But there was ambiguity in some parts of this project that made me decide how to handle some situations, what is considered an error, etc. Also I could do some things different such as use of some Regex(Regular expression) instead of checking in if statements if a token starts with or ends with something. So in the end, this project taught me Python as a language and I have more ideas about how to parse input files.