Assembler

Author : Salar_Rahnama , STU_Number: 40131850

Preface:

First of All, Hi. 👋

I Know You Know Me And I Know; Sooooo It’s Not Needed To Introduce Myself. Anyway.

In this Documentation that is for phase 1 of Assembly projects, I want to talk about :

1) The concept of the project🧠

2) Functions and Dictionaries 🚗

3)GUI(GitHub)🤷‍♂️

Concept of the project:

This project is X86 Assembler that is programmed in Python language and It has 3 sections.

First section was that we could handle transfers and operations between **Registers:**

We must define 32, 16 and 8 bit registers and assemble them to see what is behind the scene.

Second section added Memories to Handling list, and made It enjoyable.

And **Third** But not last section made us to handle Backward and Forward JMP instruction. This section was the challenging part of the project until now. We almost died in this part honestly.💔

And some common things like counters before the Assembled line and… !

Functions and Execution

This code appears to be an assembler written in Python that translates a simplified assembly language into machine code. Here's an explanation of the key components and functions:

Variables and Data Structures:

- `offset`: A variable to keep track of the current offset in the machine code.
- `labels`: A dictionary to store labels and their corresponding addresses.
- `to_print`: A list to store the generated machine code instructions.
- `jumps`: A list to store information about jump instructions that need to be filled in later.
- `append_Flag`: A flag to determine whether a space should be appended in the `to_print` list.

Functions:

1. `start()`:

- Prints a welcome message and calls `assemble_from_file()`.

2. `convertFunc(number)`, `convertFunc2(number)`, `convert16Func2(number)`, `convert16Func(number)`:

- Functions to convert binary numbers into hexadecimal strings with specific formats.

3. `assemble(instruction, first_arg, second_arg)`:

- Takes an assembly instruction and its arguments and converts it into machine code.
- Handles different cases for instructions with one or two arguments.
- Appends the generated machine code to the `to_print` list.

4. `calculate_16s_complement(negative_number)`:

- Calculates the 16's complement of a negative number.

5. `assemble_from_file()`:

- Reads the assembly code from a file (`InputTest.txt`).
- Parses each line, identifies labels, and calls `assemble()` to generate machine code.
- Updates the `offset` and handles jump instructions by storing them in the `jumps` list.

6. `jumpings()`:

- Calls `WTF_Jumps()` for each jump instruction in the `jumps` list.

7. `WTF_Jumps(index, Label2, Address)`:

- Handles jump instructions by calculating the relative jump offset and updating the `to_print` list.

8. `printings()`:

- Prints the generated machine code instructions in the `to_print` list.

Dictionaries:

- `binary_to_hex_dict`: A dictionary for converting binary digits to hexadecimal.
-

```
binary_to_hex_dict = {
    '0000': '0', '0001': '1', '0010': '2', '0011': '3', '0100': '4',
    '0101': '5', '0110': '6', '0111': '7', '1000': '8', '1001': '9',
    '1010': 'A', '1011': 'B', '1100': 'C', '1101': 'D', '1110': 'E', '1111': 'F'
}
```

- `registers_32bit`, `registers_16bit`, `registers_8bit`: Dictionaries mapping register names to binary codes.
-

```
registers_32bit = {'eax': "000", 'ebx': "011", 'ecx': '001', 'edx': '010', 'esi': '110', 'edi': '111', 'esp': '100',
                   'ebp': '101'}
registers_16bit = {'ax': "000", 'bx': "011", 'cx': '001', 'dx': '010', 'si': '110', 'di': '111', 'sp': '100',
                   'bp': '101'}
registers_8bit = {'al': '000', 'bl': '011', 'cl': '001', 'dl': '010', 'ah': '100', 'bh': '111', 'ch': '101',
                  'dh': '110'}
```

- `registers_8bit_MOD00`, `registers_16bit_MOD00`, `registers_32bit_MOD00`: Dictionaries for memory reference registers.
-

```
registers_8bit_MOD00 = {'[al]': '000', '[bl]': '011', '[cl]': '001', '[dl]': '010', '[ah]': '100', '[bh]': '111',
                        '[ch]': '101', '[dh]': '110'}
registers_16bit_MOD00 = {'[ax]': "000", '[bx]': "011", '[cx]': '001', '[dx]': '010', '[si]': '110', '[di]': '111',
                         '[sp]': '100', '[bp]': '101'}
registers_32bit_MOD00 = {'[eax]': "000", '[ebx]': "011", '[ecx]': '001', '[edx]': '010', '[esi]': '110', '[edi]': '111',
                         '[esp]': '100', '[ebp]': '101'}
```

- `instructionOpcode`: Dictionaries mapping assembly instruction mnemonics to their corresponding opcodes.
-

```
instructionOpcode = {
    'ADD': '000000', 'SUB': '001010', 'AND': '001000', 'OR': '000010', 'XOR': '001100', 'PUSH': '01010',
    'POP': '01011', 'INC': '01000', 'DEC': '01001' , 'JMP' : 'EB'
}
```

Execution:

- The script starts by calling `start()`, which initiates the assembly process.
- It reads the assembly code from the file, processes each line, and generates machine code.

- Jump instructions are processed later using the `jumps` list and the `WTF_Jumps()` function.
- Finally, it prints the generated machine code.

Please note that the code assumes a specific assembly language syntax and may not be suitable for all assembly languages.

1. **Initialization:**
    - The script starts by defining several variables, dictionaries, and lists for managing the assembly process.
    - Key dictionaries include `binary_to_hex_dict` for binary-to-hex conversion and `registers_32bit`, `registers_16bit`, `registers_8bit` for mapping register names to binary codes.
2. **Function Definitions:**
    - The script defines various functions to handle different aspects of the assembly process, such as `assemble`, `convertFunc`, `calculate_16s_complement`, etc.
    - These functions are designed to convert assembly instructions and arguments into corresponding machine code.
3. **File Reading and Processing:**
    - The `assemble_from_file` function is responsible for reading the assembly code from the "InputTest.txt" file.
    - It splits the code into lines and processes each line, extracting instructions and arguments.
    - Labels are identified and stored in the `labels` dictionary with their corresponding addresses.
    - The `assemble` function is then called to convert instructions into machine code, and the results are stored in the `to_print` list.
4. **Handling Jumps:**
    - The script identifies jump instructions during the initial assembly pass and stores information about them in the `jumps` list.

- Jump instructions are not fully resolved initially; their addresses are placeholders, as the destination addresses depend on the addresses of labels in the code.

5. **Jump Resolution:**
   - After the initial pass, the `jumpings` function is called, which, in turn, calls `WTF_Jumps` for each jump instruction.
   - The `WTF_Jumps` function calculates the relative jump offset by subtracting the current address from the destination label's address.
   - It then updates the corresponding entry in the `to_print` list with the correct jump instruction.
6. **Printing Machine Code:**
   - Finally, the `printings` function is called to print the generated machine code stored in the `to_print` list.
   - The output includes the address and corresponding machine code for each instruction.
7. **Execution Entry Point:**
   - The script begins execution with the `start` function, which initiates the assembly process by calling `assemble_from_file`.
8. **Output:**
   - The final output consists of the processed machine code instructions, including jumps with resolved addresses.

In summary, the script follows a multi-step process:

- Initialize variables, dictionaries, and lists.
- Read and process the assembly code from a file, storing labels and generating initial machine code.
- Identify and store information about jump instructions.
- Resolve jump addresses.
- Print the final machine code.

This script assumes a specific assembly language syntax and may require adaptation for different assembly languages or syntax variations.

GUI

For GUI Code there’s my repository of Github lets check it out here :

```
https://github.com/stupidtallguy/Assembler_Ph1.git
```