Assembly Project first part

Mahyar Mohammadian
40131874

1 Introduction

Abstract

The provided Python code serves as an assembly code assembler. The primary objective of this program is to convert assembly instructions into their respective hexadecimal machine code representations. This transformation is crucial for computers to execute the instructions efficiently. The code employs various dictionaries to map assembly mnemonics and operands to their binary and hexadecimal equivalents, facilitating a streamlined conversion process. Additionally, the program contains functionalities to read assembly code from a text file, process it, and output the assembled code in hexadecimal format. This documentation aims to provide clarity on the dictionaries and their roles in the overall functionality of the code.

2 Dictionaries

2.1 `binary_to_hex_dict`

This dictionary serves the purpose of converting binary numbers into their corresponding hexadecimal representations. It offers mappings for all 4-bit binary numbers, ranging from '0000' to '1111', to their respective hexadecimal counterparts, from '0' to 'F'.

Listing 1: `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'
}
```

2.2 `registers_32bit`

This dictionary represents the 32-bit registers typically found in x86 architecture. Each register mnemonic, such as 'eax', 'ebx', etc., is associated with its 3-bit binary representation.

Listing 2: `registers_32bit`

```
{
    'eax': "000",
    'ebx': "011",
    'ecx': '001',
    'edx': '010',
    'esi': '110',
    'edi': '111',
    'esp': '100',
    'ebp': '101'
}
```

2.3 registers_16bit

Similar to the 32-bit register dictionary, this one focuses on the 16-bit registers. It provides mappings of register mnemonics to their 3-bit binary representations.

Listing 3: registers_16bit

```
{
    'ax': "000",
    'bx': "011",
    'cx': '001',
    'dx': '010',
    'si': '110',
    'di': '111',
    'sp': '100',
    'bp': '101'
}
```

2.4 registers_8bit

This dictionary is dedicated to 8-bit registers. Each mnemonic is mapped to its 3-bit binary representation, representing either the lower or upper byte of certain 16-bit registers or serving specific data manipulation tasks.

Listing 4: registers_8bit

```
{
    'al': '000',
    'bl': '011',
    'cl': '001',
    'dl': '010',
    'ah': '100',
    'bh': '111',
    'ch': '101',
    'dh': '110'
}
```

subsectionregisters_32bit_MOD00 This dictionary caters to 32-bit registers in the context of ModR/M byte encoding, specifically when the 'Mod' value is '00'. It provides mappings for register mnemonics enclosed within square brackets to their respective binary representations.

Listing 5: registers_32bit_MOD00

```
{
    '[eax]': "000",
    '[ebx]': "011",
    '[ecx]': '001',
    '[edx]': '010',
    '[esi]': '110',
    '[edi]': '111',
    '[esp]': '100',
    '[ebp]': '101'
}
```

2.5 `instructionOpcode`

The `instructionOpcode` dictionary maps mnemonic instructions, such as 'ADD' or 'SUB', to their corresponding binary opcodes. This facilitates the translation of human-readable mnemonic instructions into their machine-executable binary formats.

Listing 6: `instructionOpcode`

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

3 Function Documentation

In this section, we provide documentation for the primary functions responsible for converting binary assembly code into hexadecimal representations. Each function is designed to handle different bit lengths and operand types.

3.1 `binary_to_hex(number)`

Description: Converts a 32-bit binary assembly code into a hexadecimal representation.

Parameters:

- `number`: A list containing a single string representing the 32-bit binary assembly code.

Functionality: This function utilizes a global counter to track the hexadecimal address. It breaks the binary string into four 4-bit segments and converts each segment to its corresponding hexadecimal value using the `binary_to_hex_dict`. The resultant hexadecimal value is then appended to the `printArray` list with its associated address.

```
def binary_to_hex(number):
    global counter

    hex_value = (
        binary_to_hex_dict[number[0][0:4]] +
        binary_to_hex_dict[number[0][4:8]] +
        binary_to_hex_dict[number[0][8:12]] +
        binary_to_hex_dict[number[0][12:16]]
    )
    printArray.append(["0x"+f"{counter:015d}: {hex_value}"])
    counter += 2
```

Figure 1: Visual representation illustrating the `binary_to_hex` function.

3.2 `binary_to_hex_16bit(number)`

Description: Converts a 16-bit binary assembly code into a 16-bit hexadecimal representation prefixed with '66'.

Parameters:

- `number`: A list containing a single string representing the 16-bit binary assembly code.

Functionality: Similar to the previous function, but prefixes the resultant hexadecimal value with '66' to indicate a 16-bit operation.

```
def binary_to_hex_16bit(number):
    global counter

    hex_value = '66' + (
        binary_to_hex_dict[number[0][0:4]] +
        binary_to_hex_dict[number[0][4:8]] +
        binary_to_hex_dict[number[0][8:12]] +
        binary_to_hex_dict[number[0][12:16]]
    )
    printArray.append(["0x"+f"{counter:015d}: {hex_value}"])
    counter += 3
```

Figure 2: Visual representation illustrating the `binary_to_hex_16bit` function.

3.3 `binary_to_hex_one_operand(number)`

Description: Converts a 32-bit binary assembly code for one-operand instructions into a 8-bit hexadecimal representation.

Parameters:

- `number`: A list containing a single string representing the 32-bit binary assembly code.

Functionality: This function converts the binary string into its corresponding 8-bit hexadecimal value. The resultant value is then appended to the `printArray` list with its associated address.

```
def binary_to_hex_one_operand(number):
    global counter

    hex_value = (
        binary_to_hex_dict[number[0][0:4]] +
        binary_to_hex_dict[number[0][4:8]]
    )
    printArray.append(["0x"+f"{counter:015d}: {hex_value}"])
    counter += 1
```

Figure 3: Visual representation illustrating the `binary_to_hex_one_operand` function.

3.4 `binary_to_hex_16_bit_one_operand(number)`

Description: Converts a 16-bit binary assembly code for one-operand instructions into a 16-bit hexadecimal representation prefixed with '66'.

Parameters:

- `number`: A list containing a single string representing the 16-bit binary assembly code.

Functionality: Similar to the previous function, but prefixes the resultant hexadecimal value with '66' to indicate a 16-bit operation.

```
def binary_to_hex_16_bit_one_operand(number):
    global counter

    hex_value = '66' + (
        binary_to_hex_dict[number[0][0:4]] +
        binary_to_hex_dict[number[0][4:8]]
    )
    printArray.append(["0x"+f"{counter:015d}: {hex_value}"])
    counter += 2
```

Figure 4: Visual representation illustrating the `binary_to_hex_16_bit_one_operand` function.

3.5 `binary_to_hex_8_bit_one_operand(number)`

Description: Converts an 8-bit binary assembly code for one-operand instructions into an 8-bit hexadecimal representation prefixed with 'FE'.

Parameters:

- `number`: A list containing a single string representing the 8-bit binary assembly code.

Functionality: This function converts the binary string into its corresponding 8-bit hexadecimal value prefixed with 'FE'. The resultant value is then appended to the `printArray` list with its associated address.

```
def binary_to_hex_8_bit_one_operand(number):
    global counter

    hex_value = ("FE"+
        binary_to_hex_dict[number[0][0:4]] +
        binary_to_hex_dict[number[0][4:8]]
    )
    printArray.append(["0x"+f"{counter:015d}: {hex_value}"])
    counter += 2
```

Figure 5: Visual representation illustrating the `binary_to_hex_8_bit_one_operand` function.

3.6 `main_process(instruction, first_arg, second_arg)`

Description: Processes the given assembly instruction and its arguments to produce the appropriate binary representation.

Parameters:

- `instruction`: A string representing the assembly instruction.
- `first_arg`: A string representing the first argument of the instruction.
- `second_arg`: A string representing the second argument of the instruction.

Functionality: The `main_process` function checks various conditions based on the provided instruction and its arguments. Depending on the type and combination of arguments, it constructs the corresponding binary representation of the instruction. The function then appends this binary representation to the `number` list. Once appended, the binary value is converted to its hexadecimal representation using one of the provided conversion functions.

- If both `first_arg` and `second_arg` are 32-bit registers, the function constructs the binary code and utilizes the `binary_to_hex` function.
- Conditions for 16-bit and 8-bit registers are handled similarly.
- Special cases are considered where one operand might be from `registers_32bit_MOD00`.
- If only one argument (a register) is provided, a one-operand variant of the function is used.
- For the `JMP` instruction, a special handling is done to manage jumps within the code.

Note: The function uses a global counter, `counter`, to track the position within the code during processing.

3.7 `assemble_code_from_text(code)`

Description: Processes the provided assembly code text to produce a list of instructions along with their associated addresses.

Parameters:

- `code`: A string containing the assembly code to be processed.

Functionality: The `assemble_code_from_text` function first initializes and resets various global variables, such as `counter`, `printArray`, `label_dict`, and `jmp_dict`. It then parses the provided code line-by-line, identifying instructions and their operands. Labels are handled separately, and their offsets are stored in `label_dict`.

The function contains several internal utility functions for specialized tasks. These include:

- `is_label(line)`: Identifies if a given line is a label.
- `handle_label(line)`: Processes and adds labels to `label_dict`.
- `complement16(num)`: Computes the 16-bit complement of a number.
- `JmpHandle()`: Handles jump instructions and adjusts the print array accordingly.

Once all instructions are processed, the function returns a list, `printArray`, containing the assembled code instructions along with their respective addresses.

3.8 `is_label(line)`

Description: Checks if a given line is a label.

Parameters:

- `line`: A string representing a line of code.

Returns:

- `True` if the line is a label.
- `False` otherwise.

```python
def is_label(line):
    return re.match(r'^\s*([a-zA-Z_][a-zA-Z0-9_]*)\s*:$', line)
```

Figure 6: Visual representation illustrating the `is_label` function.

3.9 `handle_label(line)`

Description: Processes a label line and stores it in the label dictionary with its corresponding offset.

Parameters:

- `line`: A string representing a line of code.

```
def handle_label(line):
    global counter
    label = re.match(r'^\s*([a-zA-Z_][a-zA-Z0-9_]*)\s*:$', line).group(1)
    label_dict[label] = counter

# iterate the file line by line for instructions
for line in lines:
    if line != '':
        if is_label(line):
            printArray.append(["0x" + f"{counter:015d}: {line} NOTHING"])
            handle_label(line)
        else:
            components = re.split(r'\s|,\s*', line)
            instructions = components[0].upper()
            first_arg = components[1] if len(components) > 1 else None
            second_arg = components[2] if len(components) > 2 else None

            main_process(instructions, first_arg, second_arg)
```

Figure 7: Visual representation illustrating the `handle_label` function.

3.10 `complement16(num)`

Description: Computes the 16-bit complement of a given hexadecimal number.

Parameters:

- `num`: A string representing a hexadecimal number.

Returns:

- The 16-bit complement of the input number.

```python
def complement16(num):
    hexnum = [*num]
    for i in range(len(hexnum)):
        hexnum[i] = hex(15 - int(hexnum[i], 16))[2:]
    temp = "".join(hexnum)
    if len(hexnum) < 2:
        temp = "f" + temp
    return hex((int(temp, 16)) + 1)[2:]
```

Figure 8: Visual representation illustrating the `complement16` function.

3.11 `JmpHandle()`

Description: Adjusts jump instructions in the print array based on the computed offsets.

Functionality: This function iterates through the jump dictionary and the label dictionary to compute the correct jump offsets. It then modifies the print array to reflect these corrected offsets.

```python
def JmpHandle():
    temp = []
    index = 0
    for j in jmp_dict:
        for i in label_dict:
            if j == i:
                temp.append(label_dict[i] - (jmp_dict[j] + 2))

        for i in range(len(printArray)):
            if not printArray[i]:
                if temp[index] >= 0:
                    printArray[i] = ["0x" + f"{jmp_dict[j]:015d}: EB 0{temp[index]}"]
                else:
                    temp[index] = str(complement16(hex(int(temp[index]))[3:])).upper()
                    printArray[i] = ["0x" + f"{jmp_dict[j]:015d}: EB {temp[index]}"]
                index += 1
                break
```

Figure 9: Visual representation illustrating the `JmpHandle` function.

3.12 `load_file_data()`

Description: Loads data from a selected text file and displays it in a specified text widget.

Functionality: The `load_file_data` function prompts the user to select a text file via a file dialog. Once a file is selected, it reads the content of the file and inserts it into a specified text widget, `file_data_text`.

File Dialog: The function utilizes the `filedialog` module to display a file dialog window. Users can navigate their file system and choose a text file to load.

Parameters: None.

Returns: None.

```
def load_file_data():
    file_path = filedialog.askopenfilename(title="Select Assembly File", filetypes=[("Text files", "*.txt")])
    if file_path:
        with open(file_path, 'r') as file:
            file_data = file.read()
        file_data_text.delete(1.0, tk.END)
        file_data_text.insert(tk.END, file_data)
```

Figure 10: Visual representation illustrating the `load_file_data` function.

3.13 `assemble_loaded_file()`

Description: Processes the loaded data from a text widget to produce assembled code and displays it in another specified text widget.

Functionality: The `assemble_loaded_file` function retrieves the content from the `file_data_text` widget, which is presumably loaded with assembly code data. If the data contains valid content, the function attempts to assemble the code using the `assemble_code_from_text` function. The resulting assembled code is then inserted into the `assembled_code_text` widget, replacing any previous content.

If there's an error during the assembly process, the function displays the error message in the `assembled_code_text` widget. If no data is present in the `file_data_text` widget, a message indicating the absence of data is displayed.

Parameters: None.

Returns: None.

```
def assemble_loaded_file():
    file_data = file_data_text.get(1.0, tk.END)
    if file_data.strip():
        try:
            assembled_code = assemble_code_from_text(file_data)
            assembled_code_text.delete(1.0, tk.END)
            for line in assembled_code:
                assembled_code_text.insert(tk.END, line[0] + '\n')
        except Exception as e:
            assembled_code_text.delete(1.0, tk.END)
            assembled_code_text.insert(tk.END, f"Error: {str(e)}")
    else:
        assembled_code_text.delete(1.0, tk.END)
        assembled_code_text.insert(tk.END, "No data to assemble.")
```

Figure 11: Visual representation illustrating the `assemble_loaded_file` function.

4 Graphical User Interface (GUI)

4.1 Main GUI Window Configuration

The main graphical user interface (GUI) window for the Assembly Code Assembler application is constructed using the Tkinter library in Python. The window is titled "Assembly Code Assembler" and uses a custom icon for its representation. The background color of the window is set to a dark shade (`#282c34`).

4.2 Widgets and Styling

The application incorporates various widgets, including buttons and text areas, styled to provide a cohesive look and feel.

- **Buttons:** Two buttons are present:
  - `Load File` - Initiates the process of loading assembly code from a text file.
  - `Assemble` - Initiates the assembly process for the loaded code.

  These buttons utilize the `ttk` module for a more refined appearance with custom styling, including color, padding, and font.
- **Text Areas:** Two text areas are present:
  - `file_data_text` - Displays the loaded assembly code from the selected file. This text area is situated within a light gray frame.
  - `assembled_code_text` - Displays the assembled code after the assembly process. This text area is also within a light gray frame.

The overall GUI employs a custom font (`Helvetica`, size 12) to enhance readability and maintain a consistent design language.

4.3 Styling Details

The styling of the GUI components, particularly the buttons, is achieved using the `ttk.Style()` class. Buttons are given a flat relief, with specific background and foreground colors to ensure visibility and aesthetics.

4.4 Execution

Upon setting up the GUI components, the main event loop is invoked using the `root.mainloop()` method, ensuring the application remains responsive to user interactions.

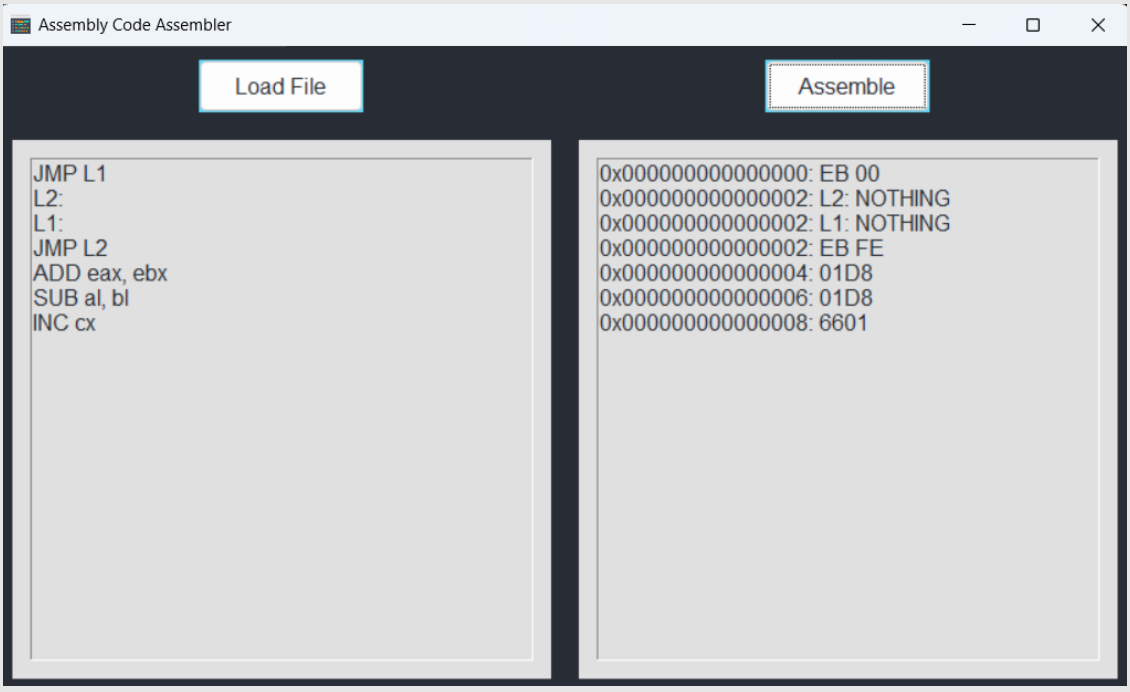

Figure 12: Visual representation illustrating the `Graphical User Interface` .