Assembly Language Course

X86 ASSEMBLER

Parsa HaghighatGoo

Professor: Dr. Esfandiari

semester Fall - 2023

0.1 Introduction

In this project, we designed an x86 Online Assembler using Python for the assembler, the Flask framework for the backend, and created the UI with HTML/CSS/-Bootstrap and JavaScript.
This project also includes a GUI version and an executable (exe) file. You can find all the project files on [GitHub](https://github.com/ParsaHaghighatgoo/assembler-x86).

Contents

0.1 Introduction 1

Contents **2**

1 Assembler Overview **4**

1.1 Assembly Dictionaries 5

1.1.1 32-bit Registers 5

1.1.2 32-bit Memory Operands 5

1.1.3 16-bit Registers 6

1.1.4 16-bit Memory Operands 7

1.1.5 8-bit Registers 7

1.1.6 8-bit Memory Operands 8

1.1.7 Instruction Opcodes 8

1.2 Complement16 Function 9

1.3 Find Counter Address Function 9

1.4 Assembler Function 11

1.5 Jump (jmp) Function 14

1.5.1 14

1.5.2 Parameters: 14

1.5.3 Description: 14

1.5.4 Code: 15

1.6 File Reading 16

1.6.1 Python Code 16

1.6.2 Description 16

1.7 Initialization 16

1.7.1 Python Code 16

1.7.2 Description 16

1.8 Code Processing 17

1.8.1 Python Code 17

1.8.2 Description 17

1.9 Jump Processing 18

1.9.1 Python Code 18

1.9.2 Description 18

1.10 Print Output 18

1.10.1 Python Code 18
1.10.2 Description 18

2 GUI Input **19**
2.1 Python Code 19
2.2 Description 19
2.3 Assemble Code 19
2.4 Python Code 19
2.5 Description 20
2.6 EasyGui 21

3 Exe file **22**

4 Flask Application **22**
4.1 Flask App Script 22
4.2 Description 24

5 Web Application **25**

6 Thanks **26**

1 Assembler Overview

This Python-based x86 assembler supports 32-bit assembly instructions. The assembler translates human-readable assembly code into machine code, facilitating the execution of programs on 32-bit architectures.

1.1 Assembly Dictionaries

In the context of the x86 assembler, several dictionaries are defined to facilitate the translation of assembly mnemonics to their corresponding binary representations. These dictionaries play a crucial role in encoding various components of assembly language, including registers, memory operands, and instruction opcodes.

1.1.1 32-bit Registers

Listing 1.1: 32-bit Registers

```
registers_32bit = {
    "eax": "000",
    "ecx": "001",
    "edx": "010",
    "ebx": "011",
    "esp": "100",
    "esi": "110",
    "edi": "111",
    "ebp": "101"
}
```

1.1.2 32-bit Memory Operands

Listing 1.2: 32-bit Memory Operands

```
memory_32bit = {
    "[eax]": "000",
    "[ecx]": "001",
    "[edx]": "010",
    "[ebx]": "011",
    "[esp]": "100",
    "[esi]": "110",
    "[edi]": "111",
    "[ebp]": "101"
}
```

1.1.3 16-bit Registers

Listing 1.3: 16-bit Registers

```
registers_16bit = {
    "ax": "000",
    "cx": "001",
    "dx": "010",
    "bx": "011",
    "sp": "100",
    "si": "110",
    "di": "111",
    "bp": "101"
}
```

1.1.4 16-bit Memory Operands

Listing 1.4: 16-bit Memory Operands

```
memory_16bit = {
    "[ax]": "000",
    "[cx]": "001",
    "[dx]": "010",
    "[bx]": "011",
    "[sp]": "100",
    "[si]": "110",
    "[di]": "111",
    "[bp]": "101"
}
```

1.1.5 8-bit Registers

Listing 1.5: 8-bit Registers

```
registers_8bit = {
    "al": "000",
    "cl": "001",
    "dl": "010",
    "bl": "011",
    "ah": "100",
    "ch": "101",
    "dh": "110",
    "bh": "111"
}
```

1.1.6 8-bit Memory Operands

Listing 1.6: 8-bit Memory Operands

```
memory_8bit = {
    "[al]": "000",
    "[cl]": "001",
    "[dl]": "010",
    "[bl]": "011",
    "[ah]": "100",
    "[ch]": "101",
    "[dh]": "110",
    "[bh]": "111"
}
```

1.1.7 Instruction Opcodes

Listing 1.7: Instruction Opcodes

```
InstructionOpCode = {
    "add": "000000",
    "sub": "001010",
    "and": "001000",
    "or": "000010",
    "xor": "001100",
    "push": "01010",
    "pop": "01011",
    "inc": "01000",
    "dec": "01001"
}
```

1.2 Complement16 Function

The `complement16` function takes a hexadecimal number as input and computes its 16's complement. Here is the Python code for the function:

Listing 1.8: Python code for complement16 function

```
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:]
```

1.3 Find Counter Address Function

The `findCounterAddress` function takes a string (`output`) as input and finds the counter address based on a specific condition. Here is the Python code for the function:

Listing 1.9: Python code for findCounterAddress function

```
def findCounterAddress(output):
    index = 0
    for i in range(len(output)):
        if (output[i]) == ":":
            index = i
            break
    index += 2
    out = ""
    for i in range(index, len(output)):
```

```
10          out += output[i]
11      cnt = len(out.split(" "))
12      return cnt
```

1.4 Assembler Function

The assembler function is a versatile assembly code generator that translates high-level assembly instructions into machine code. It supports a variety of instructions and addressing modes, allowing for flexibility in coding x86 assembly.
Register-Register Instructions This section handles instructions where both operands are 32-bit registers. It generates the corresponding machine code, appends it to the output, and updates the program counter.
Register-Memory 32-bit For instructions involving a 32-bit register and a memory operand, this section produces the corresponding machine code. It considers different cases and updates the program counter accordingly.
Jump and Other Special Conditions Handling special instructions like jumps, this section appends an empty output for further processing and maintains a record of jump instructions.
Other Instruction Scenarios This section deals with various instruction scenarios, such as pushing values onto the stack, memory operations, and error handling for invalid instructions.
The assembler function demonstrates a modular and organized approach to generating assembly code. It can be extended to support additional instructions and addressing modes as needed.

Listing 1.10: Assembler Function

```
def assembler(instruction, first_arg, second_arg,
    addressCounter):
    number = []
    index = 0
    output = ""

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Register-Register Instructions %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    if first_arg in registers_32bit and second_arg in
        registers_32bit:
        output += instruction.upper() + " " +
            first_arg + "," + second_arg + " <====> "
        output += "0x"
        number.append(
            InstructionOpCode[instruction] + "01" +
                '11' + registers_32bit[second_arg] +
                registers_32bit[first_arg])
        first_print = hex(int(number[index][:8], 2))
            [2:]
        ...

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Register-Memory 32-bit    %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    elif first_arg in registers_32bit and second_arg
        in memory_32bit:
        output += instructions.upper() + " " +
            first_arg + "," + second_arg + " <====> "
        output += "0x"
        number.append(
            InstructionOpCode[instruction] + "11" +
                '00' + registers_32bit[first_arg] +
```

```
                        memory_32bit[second_arg])
            ...

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Jump and Other Special Conditions %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    elif instruction == "jmp":
        output = ""
        printed.append(output.upper())
        jmps.append([first_arg, addressCounter, len(
            printed) - 1])
        ...

    elif ((int(first_arg)) <= 127) and (int(first_arg)
        >= -128) and instruction == "push":
        output += instructions.upper() + " " +
            first_arg + " <====> "
        output += "0x"
        ...

    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    % Handle Other Instruction Scenarios %
    %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    else:
        print("Invalid Instruction or reg/m/imm")
        return 0
```

The function works the same for other inputs. If you're interested, you can check the code on my GitHub; I've included the link in the introduction section. I aim to elucidate each segment of the code. Take the reg/reg32 section, for instance; within this function, I initially construct a string comprising the input instruction and the function's offset (generated using the findAddressCounter function, as detailed in its corresponding section). Subsequently, I create the opcode utilizing a pre-defined dictionary called diserd, convert the binary opcode of the register to hexadecimal, finalize the output string, and ultimately append it to a global list named 'printed' upon completion of the process.
The function works the same for other inputs.

1.5 Jump (jmp) Function

The `jmp` function handles the assembly instruction `jmp`, facilitating an unconditional jump to a specified label in the code. It calculates the relative offset required for the jump and assembles the corresponding machine code. The result is then formatted into an output string, including the updated program counter value.

1.5.1

sectionFunction Signature:

```
def jmp(first_arg, addressCounter, index):
```

1.5.2 Parameters:

- `first_arg`: The target label for the jump.
- `addressCounter`: The current address counter value.
- `index`: Index used for updating the global list.

1.5.3 Description:

The function calculates the relative offset by subtracting the label address from the current address counter, then converts it to machine code. The output string includes the assembled instruction, label, and the updated program counter value. The result is stored in the global list `printed`.

1.5.4 Code:

Listing 1.11: Assembler Function

```
def jmp(first_arg, addressCounter, index):
    output = ""
    number = []
    output += "jmp" + " " + first_arg + " <====> "
    output += "0x"
    labelAddress = labels[first_arg]
    processedNum = labelAddress - (addressCounter + 2)
    if processedNum < 0:
        first_print = str(complement16(hex(int(
            processedNum))[3:]))
    else:
        first_print = hex(int(processedNum))[2:]
    number.append(first_print)
    output += ("0" * (16 - len(str(addressCounter))))
      + str(addressCounter)
    output += ": eb "
    if len(first_print) == 1:
        output += "0" + first_print
    else:
        output += first_print

    printed[index] = output.upper()
    return
```

1.6 File Reading

1.6.1 Python Code

Listing 1.12: File Reading

```
file_path = "AssemblyProject1.txt"

with open(file_path, 'r') as file:
    code = file.read()

lines = code.splitlines()
```

1.6.2 Description

This section reads the contents of a file named `AssemblyProject1.txt` and splits the code into lines for further processing.

1.7 Initialization

1.7.1 Python Code

Listing 1.13: Initialization

```
addressCounter = 0
labels = {}
jmps = []
printed = []
```

1.7.2 Description

These variables are initialized to keep track of the address counter, labels, jump instructions, and printed outputs.

1.8 Code Processing

1.8.1 Python Code

Listing 1.14: Code Processing

```
try:
    for line in lines:
        splitT = line.split(" ")
        instructions = splitT[0].lower()
        if ":" in line:
            first_arg = None
            second_arg = None
        else:
            regs = splitT[1].split(",")
            if len(regs) == 2:
                first_arg = regs[0].lower()
                second_arg = regs[1].lower()
            elif len(regs) == 1:
                first_arg = regs[0].lower()
                second_arg = None

        counterAdding = assembler(instructions,
           first_arg, second_arg, addressCounter)
        addressCounter += counterAdding
        if ":" in instructions:
            labels[instructions[0:len(instructions) -
               1]] = addressCounter
except:
    print("Invalid input")
```

1.8.2 Description

This section processes each line of the code, determines the instruction and its arguments, and updates the address counter. Labels are also identified and stored.

1.9 Jump Processing

1.9.1 Python Code

Listing 1.15: Jump Processing

```
for i in jmps:
    jmp(i[0], i[1], i[2])
```

1.9.2 Description

This part of the code processes jump instructions using the `jmp` function.

1.10 Print Output

1.10.1 Python Code

Listing 1.16: Print Output

```
for i in printed:
    print(i)
```

1.10.2 Description

This section prints the assembled output.

2 GUI Input

2.1 Python Code

Listing 2.1: GUI Input

```
import easygui

def main(addressCounter=0):
    msg = "Enter your x86 assembly code:"
    title = "x86 Assembler"

    # Use codebox for multi-line input
    user_input = easygui.codebox(msg, title, "")
```

2.2 Description

This section displays a GUI input box using EasyGUI's `codebox` function, allowing the user to enter x86 assembly code.

2.3 Assemble Code

2.4 Python Code

Listing 2.2: Assemble Code

```
1     if user_input:
2         output_file_path = easygui.filesavebox("Save
              Input to File", "x86 Assembler", filetypes
              =["*.txt"])
3         output = "\n".join(printed)
4         easygui.msgbox("Assembled Code:\n\n{}".format(
              output), "Assembler Output")
5     else:
6         easygui.msgbox("Output file not selected.", "
              x86 Assembler")
```

2.5 Description

If the user provides input, this section prompts the user to save the input to a file. It then assembles the code, joins the printed output, and displays it in a message box. If no input is provided, it shows a message box indicating that the output file is not selected.

2.6 EasyGui

"I created this GUI using EasyGUI, and here are some photos of it."

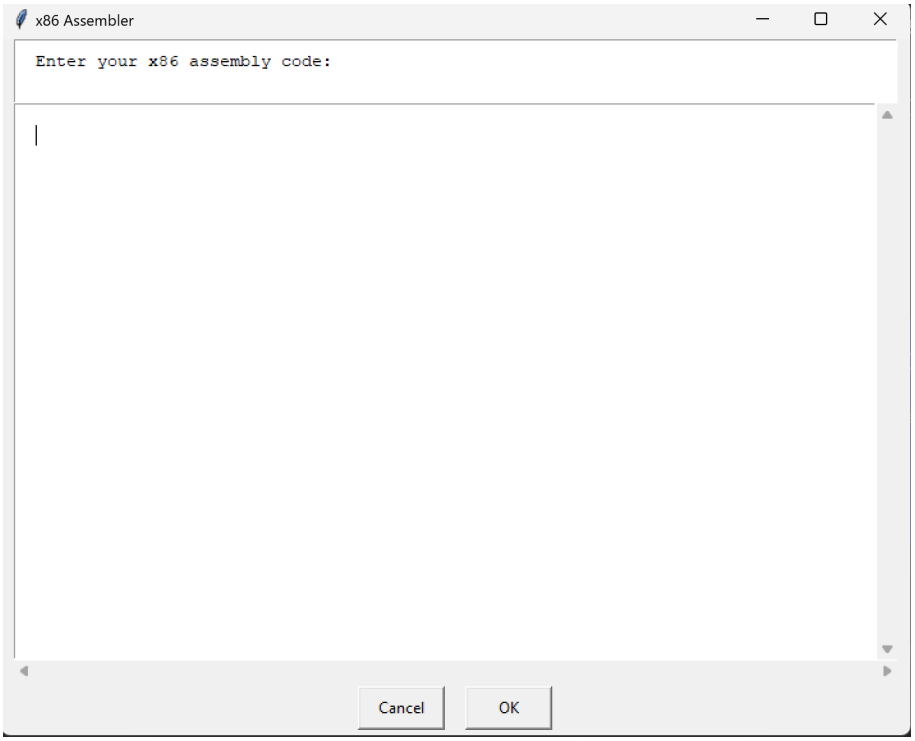

Figure 1: Input

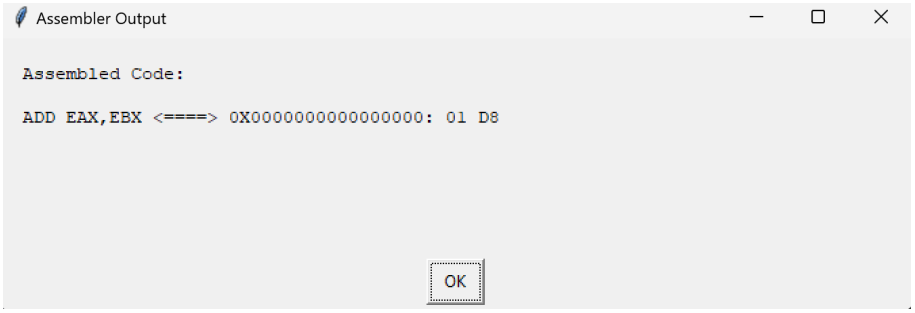

Figure 2: Output

3 Exe file

I created the executable file for this project using CxFreeze, and here is its code

Listing 3.1: Exe Setup

```
from cx_Freeze import setup, Executable
setup(
    name="x86assembler",
    version="1.0",
    description="this is a x86 assembler.",
    executables=[Executable("x86assembler.py")],
)
```

4 Flask Application

In this section, we provide an overview of a Flask application that serves as an interface for running x86 assembler scripts. The application includes routes for different assemblers and an endpoint for executing the assembler script.

4.1 Flask App Script

Listing 4.1: Flask App Script

```
from flask import Flask, render_template, request,
    jsonify, url_for
import subprocess

app = Flask(__name__)

@app.route('/')
def index():
    return render_template('index.html')

```

```
10      @app.route('/parsa')
11      def parsaAssembler():
12          return render_template('indexp.html')
13
14      @app.route('/sina')
15      def sinaAssembler():
16          return render_template('indexs.html')
17
18      @app.route('/assemblerVS', methods=['POST'])
19      def run_sinascript():
20          data = request.get_json()
21          input_data = data['input']
22          with open('AssemblyProject1.txt', 'w') as f:
23              f.write(input_data)
24          result = subprocess.check_output(['python', '
               assembler.py', input_data],
               universal_newlines=True)
25          return jsonify({'output': result})
26
27      @app.route('/assemblerVP', methods=['POST'])
28      def run_script():
29          data = request.json
30          input_data = data['input']
31          with open('AssemblyProject1.txt', 'w') as f:
32              f.write(input_data)
33
34          # Execute your Python script with the input
35          result = subprocess.check_output(['python', '
               assembler.py', input_data],
               universal_newlines=True)
36
37          return jsonify({'output': result})
38
39      if __name__ == '__main__':
```

```
40          app.run(debug=True)
```

4.2 Description

This Flask application script defines routes for different assemblers and provides an endpoint for executing the assembler script. The routes (`/`, `/parsa`, `/sina`) render corresponding HTML templates. The `/assemblerVS` and `/assemblerVP` endpoints handle POST requests, write the input data to a file, execute the assembler script, and return the output.

5 Web Application

If you want to see it, you should run the Flask server on your local host; it doesn't live yet (as of 12/20/23). I will host this web app, and it will go live later. Here are some photos of it. It includes some animations. If you want to see them, you should run the server.

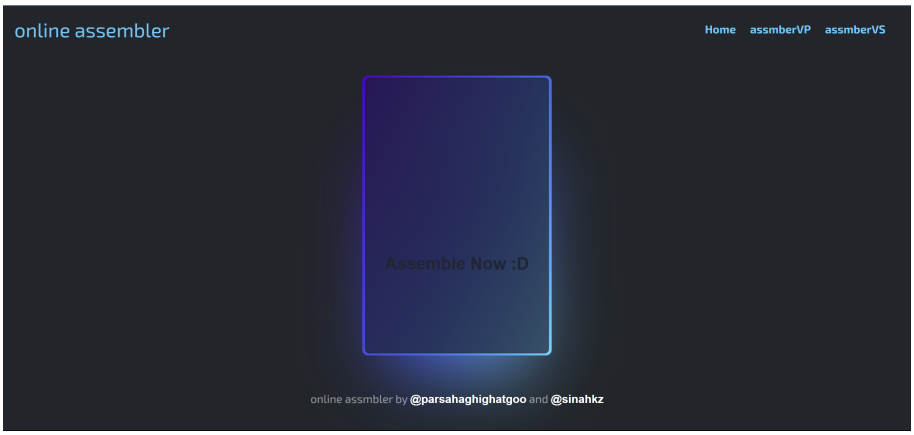

Figure 3: Home Page

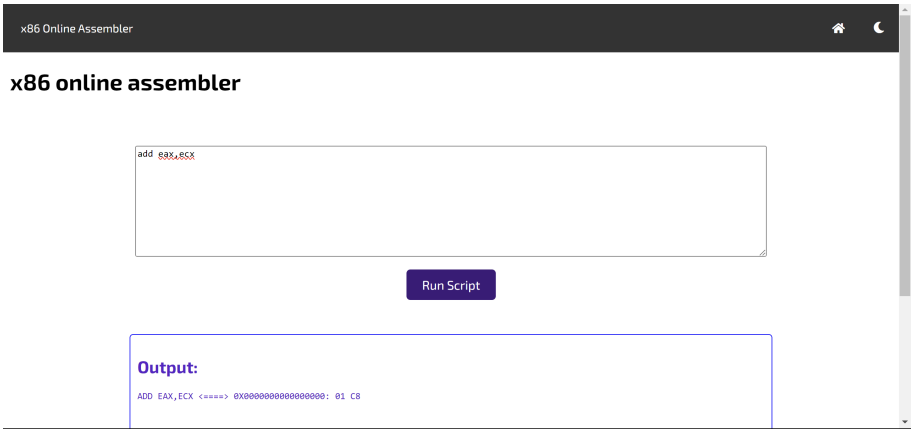

Figure 4: AssemblerVP

6 Thanks

I trust that this document proves beneficial to you.
I appreciate your time and consideration. If you have any further questions or need additional assistance, please feel free to reach out. Thank you for using my document.