Assembly Project Phase 2

> This Project is coded and this documentation is compiled by **Amirali Kazerooni**

Description

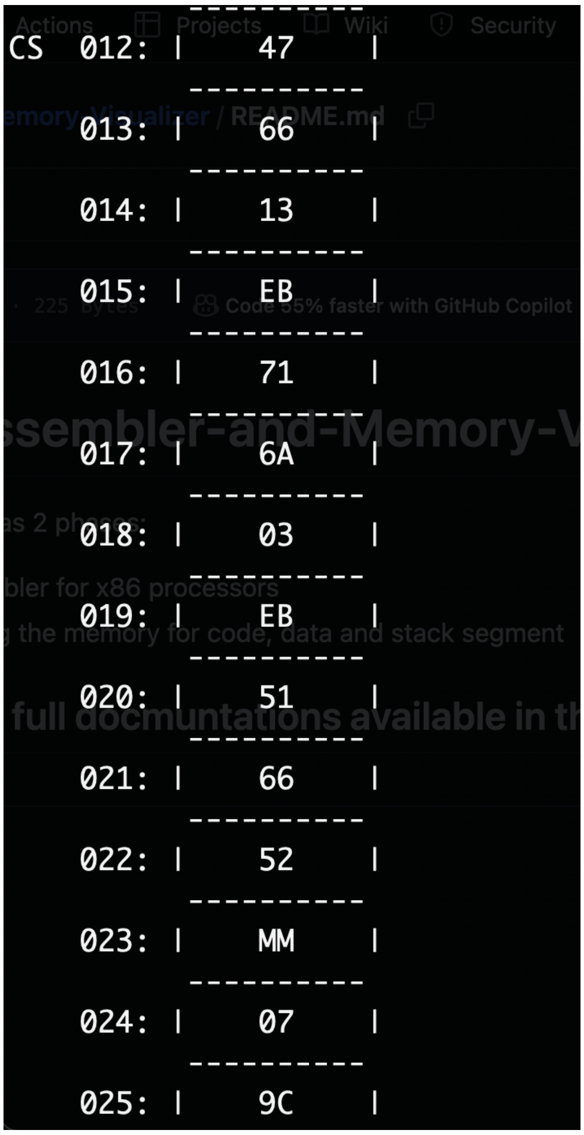
This Project which is written in python takes an assembly code like below for **x86** architecture as input and visualizes the memory of a microcontroller that has **256** bytes of memory and is running the code

Visualization

The memory is broken into three segments: **code, stack and data segment;** and each of them occupy only **32 bytes** of memory which is guaranteed that an input file that makes them **overlap** won't be given to the program

The visualization is per below:

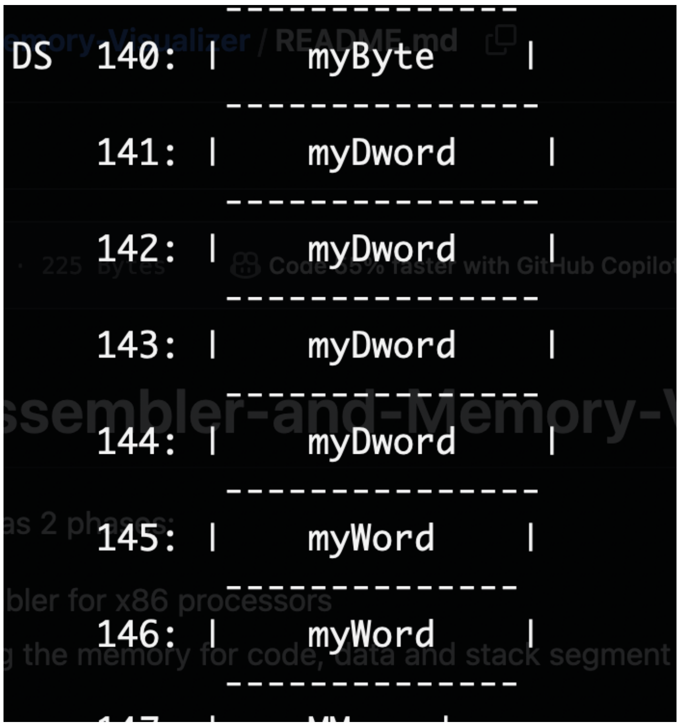
- The 32 BYTE segments that are allocated for each of the three segments should be filled with **"MM"** so that we know these are for either code or stack or data segment and the rest of the unused blocks of memory should be filled with **"XX"**
- In each block of **code segment** we should have **machine codes** of instructions in **little endian** order and **the starting memory offset of each instruction should be even** and if not a **"MM"** should be added to make it even



```
               ----------
CS  012: |     47     |
               ----------
    013: |     66     |
               ----------
    014: |     13     |
               ----------
    015: |     EB     |
               ----------
    016: |     71     |
               ----------
    017: |     6A     |
               ----------
    018: |     03     |
               ----------
    019: |     EB     |
               ----------
    020: |     51     |
               ----------
    021: |     66     |
               ----------
    022: |     52     |
               ----------
    023: |     MM     |
               ----------
    024: |     07     |
               ----------
    025: |     9C     |
```

code segment example

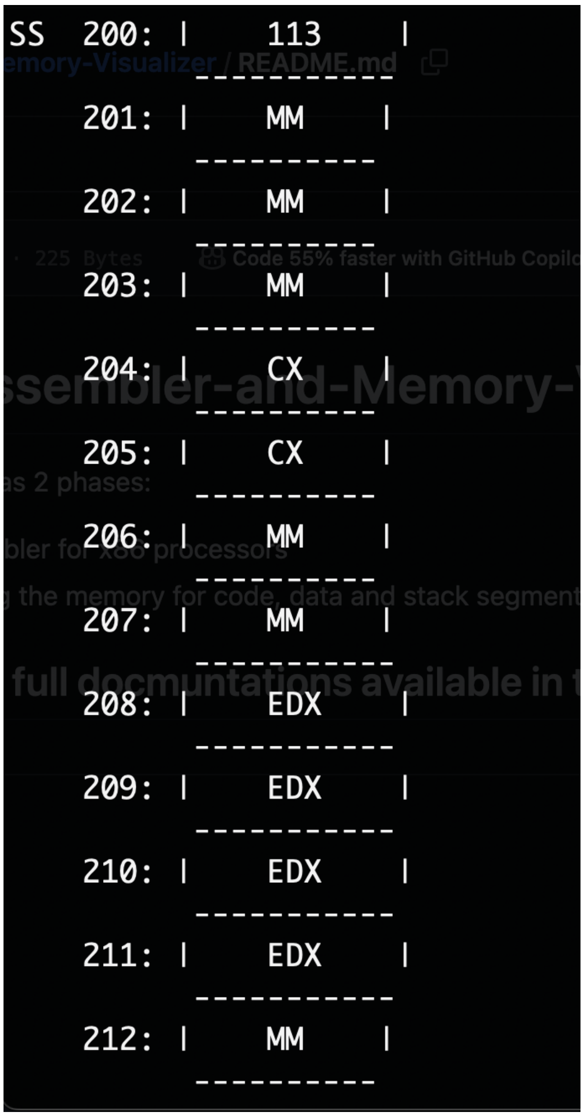
- The **data segment** should be filled with the names of variables that are defined in data segment of the code and each of them should occupy some number of memory blocks based on their size (**BYTE, WORD, DWORD**)



```
              --------------
DS   140: |     myByte     |
              ---------------
     141: |     myDword     |
              ---------------
     142: |     myDword     |
              ---------------
     143: |     myDword     |
              ---------------
     144: |     myDword     |
              ---------------
     145: |     myWord     |
              --------------
     146: |     myWord     |
              --------------
```

data segment example

- The **stack segment** should be the last state of the stack meaning if 3 things are pushed and then one poped, only the first two gets printed.
    - Note that the values PUSHed should be 32 bit so if an 8 bit number is pushed then 4 blocks of memory should be occupied but only the first one should have the value of that 8 bit number



```
SS  200: |    113    |
          -----------
    201: |    MM    |
          ----------
    202: |    MM    |
          ----------
    203: |    MM    |
          ----------
    204: |    CX    |
          ----------
    205: |    CX    |
          ----------
    206: |    MM    |
          ----------
    207: |    MM    |
          -----------
    208: |    EDX    |
          -----------
    209: |    EDX    |
          -----------
    210: |    EDX    |
          -----------
    211: |    EDX    |
          -----------
    212: |    MM    |
          ----------
```

Stack segment example

- Labels that are used meaning a JMP was done to them should get printed in the data segment
- Memory offset of each block should be printed and the starting offset of each segment should be marked by **CS, DS and SS**

Input file structure:

An example of input file is like below; note that in the parentheses of each segment, **the starting memory offset of that segment** is written which guaranteed that is an **even number**

```
.stack(200)

.data(140)
```

```
    myByte byte ?
    myDword Dword 1212
    myWord WORD 22
.code(12)
    bOos:
    INC di
    JMP first
    PUSH 113
    jmp myLabel
    PUSH cx
    PUSH edx
    myLabel:
    PUSH 127723727
    AND ax, si
    bashe:
    jMp bOos
    inc eax
    POp eax
```

Flow of the program

First be aware that the **machine codes** are being calculated via the code used in the **phase 1 of the project** which was an assembler and in this phase, some small parts of the phase 1 code has been changed and some code is added to it

Below is the description for added functions:

Main2()

The program starts here:

📌 First `read__and_fill_data()` functions is called which reads the input file and extract the variables that are defined in the data segment and fill the `dataToPrint` list based on the explanations of the visualization section and after that the `read_and_add_labelnames()` function gets called which adds the used lable names to **dataToPrint** list

The next thing that is done in this function is that the input file is read and the memory offset of each segment is extracted and saved in **codeIndex, stackIndex and dataIndex**

Then the `main()` function gets called which is basically the code for the phase 1 of the project with the code being the code in the code segment of the input file and here(in the main function) we have some changes or added code:

1. When the instructions is **PUSH**, the destination operand is added to `stackToPrint` list based on the explanations of stack segment and when the instruction is **POP**, the last 4 values of **stackToPrint** gets deleted
2. There are some changes that handle the adding of **machine codes** in the **little endian order** to the `codeToPrint` list and the **even memory offset** for starting of each instruction when the machine code is calculated for each line of the code

Lastly the `makeFinal()` and `printFinal()` functions get called in **main2()** which handle the making of the `finalPrint` list which is initialized with **256 “XX”s** and printing that final list based on the explanations in the visualization section when prepared