ELE 402 - GRADUATION PROJECT II
FINAL REPORT

HACETTEPE UNIVERSITY
DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING

PROJECT TITLE: DEVELOPMENT OF PENTIUM MICROPROCESSOR EMULATOR EMUPENT

PROJECT GROUP MEMBERS:
Özgür TANRIVERDİ 21828882
Doğukan DOĞRUBUDAK 21828437

PROJECT SUPERVISOR: PROF. DR. UĞUR BAYSAL

SUBMISSION DATE: 07.06.2024

SPRING 2023-2024

ABSTRACT

This report outlines the design and development of EMUPENT which is a Pentium microprocessor emulator. EMUPENT replicates the functionalities of a Pentium microprocessor using modern software techniques, focusing on accurately emulating its architecture and instruction set. Implemented in Python with a graphical user interface developed using the Tkinter library, EMUPENT supports essential features such as arithmetic and logic operations, handling interrupts, and implementing crucial flags. The project involved multiple iterations, with continuous testing and refinement to ensure accuracy and performance. This report details the project's objectives, design process, methods used, and final outcomes, highlighting both the strengths and limitations of the EMUPENT emulator.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

1. INTRODUCTION

This report provides a comprehensive overview of the development of EMUPENT, a Pentium microprocessor emulator. The project description section offers a detailed explanation of the Pentium microprocessor, its architecture, and its features, such as superscalar execution, separate data and instruction caches, and bus cycle pipelining. The engineering standards and design constraints section discusses the relevant engineering standards and constraints considered during the design and implementation of EMUPENT.

The sustainable development goals section analyzes how the project aligns with specific sustainable development goals. The background section provides information on the background knowledge acquired from previous coursework and additional research conducted for the project. The methods section describes the methods and techniques used in the development of EMUPENT, including the tools and programming languages employed.

The preliminary design section outlines the initial design concepts and prototypes developed during the early stages of the project. The design process section gives a detailed account of the iterative design process, including the various stages of development, testing, and evaluation. The testing and results section presents the testing methodologies used and the results obtained, demonstrating the emulator's functionality and performance.

The final design section offers a comprehensive description of the final design, including the implemented features, design choices, and the reasoning behind them. The meeting the constraints and engineering standards section assesses how the final design meets the identified constraints and adheres to engineering standards. The cost analysis section provides a breakdown of the costs involved in the development of EMUPENT and an evaluation of its cost-effectiveness.

The team work section gives insights into the collaboration and teamwork dynamics that contributed to the project's success.

Finally, the comments and conclusions section includes final remarks on the project's outcomes, evaluating its strengths and weaknesses, and suggesting potential future improvements. This structured approach ensures a thorough presentation of the project, providing readers with a clear understanding of the development process and the final results of the EMUPENT emulator.

2. PROJECT DESCRIPTION

In order to design our project, we first need to understand the features andarchitecture of the Pentium processor.

Pentium Microprocessor is one of the powerful family members of Intel’s X86 microprocessor. It is an advanced superscalar 32-bit microprocessor, introduced in theyear 1993 that contains around 3.1 million transistors.

It has a 64-bit data bus and a 32-bit address bus that offers 4 Gb of physical memoryspace. While the maximum clock rating offered is around 60 to 233 MHz.

The architectural representation of the Pentium processor is considered to be an advancement of 80386 and 80486 microprocessors. Basically, Pentium has included modifications related to cache structure, the width of the data bus, numeric coprocessor with faster speed along with providing dual integer processor.

In the case of a Pentium processor, there are two caches, one for caching data while another for caching information and each one is of 8K size. By using a dual integer processor, two instructions can be executed in each clock cycle. The data bus width in Pentium is 64-bit which was 32-bit in 80386 and the numeric coprocessor exhibits quite a faster speed than that of 80486.

2.1 FEATURES OF PENTIUM PROCESSOR

Pentium processor offers some features. This features are superscalar architecture, separate data and instruction caches, bus cycle pipelining, execution tracing, 64-bit data bus, internal parity checking, dynamic branch prediction, dual processing supportand performance monitoring.

We have already mentioned in the beginning that Pentium is a superscalar processor.

Superscalar Processors:

A special category of microprocessors that involves a parallel approach for instruction execution called instruction-level parallelism through which more than one instructiongets executed in one clock cycle is called superscalar processors. It is famous as a second-generation RISC processor because RISC is the ones that operate in a faster manner with reduced instruction sets.
Unlike scalar processors that have the ability to execute maximal one instruction perclock cycle, the superscalar processor uses the approach of simultaneously executingtwo instructions in one clock cycle. The superscalar processors perform this task by sending multiple instructions to various execution units at the same time. Hence thisprovides high throughput.

2.2 Architecture of Pentium Microprocessor

The figure 2.2 given below is the architectural representation of Pentium Processor.

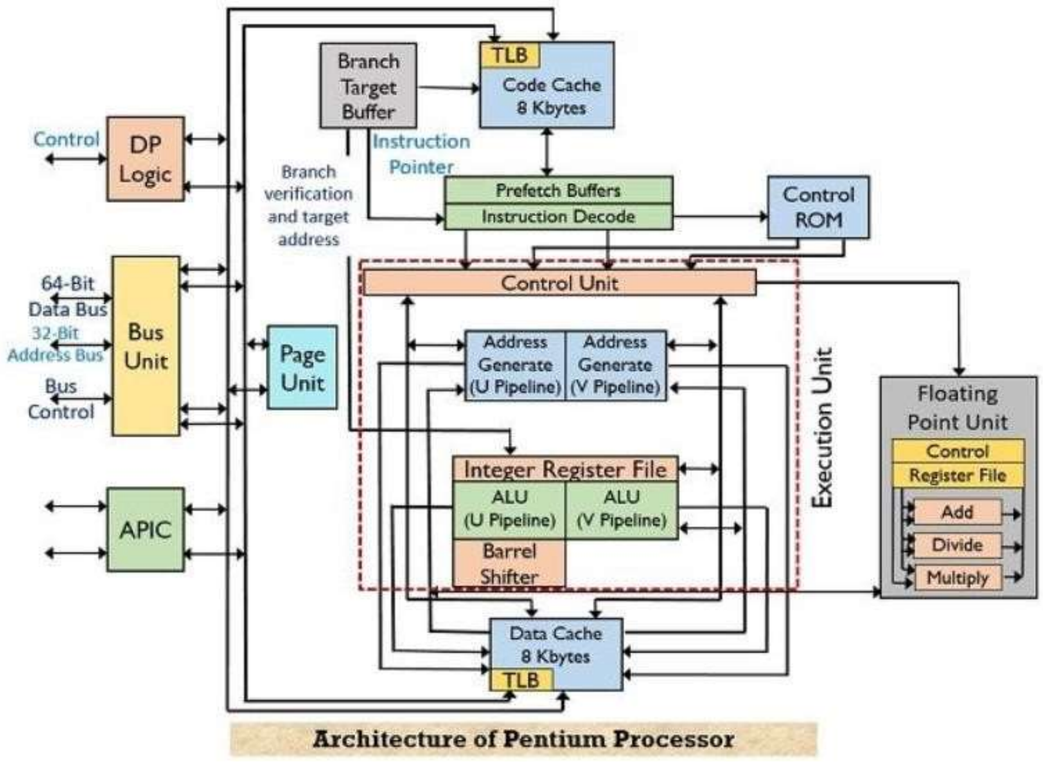


Figure 2.1. Architecture of Pentium

The various functional units are bus unit, paging unit, control ROM, prefetch buffer, execution unit with two integer pipeline (U-pipe and V-pipe), code cache, data cache, ınstruction decode, branch target buffer, dual processing logic and advanced programmable interrupt controller.

Let us now analyze, how the architectural operation takes place.

The bus unit of the architecture sends the control signal and fetches code and data from external memory and IO devices. The size of the external data bus is 64-bit through which burst read and burst write-back cycles can be achieved. The paging unitin the architecture provides optional extensions of around 2 to 4 Mb page sizes.

In order to load the instructions into the execution unit, code cache, branch target buffer and prefetch buffers operate together. The code cache or the external memory holds the instructions from where these are fetched. While the branch target buffer holds the address of the respective branch and the TLB (translational lookaside buffer) within the code cache converts the linear address into the physical address that is usedby the code cache.

This processor contains pairs of prefetch buffers having a size of 32-byte that combinedly operate with branch target buffer. Both the buffers operate independentlybut not at the same time. One of the prefetch buffers starts fetching the instructions ina sequential manner till the time branch instruction has not occurred. However, as soon as the branch instruction is fetched by the prefetch buffer then BTB (branc targetbuffer) will check for the branch but once it is checked by BTB that branch has not occurred then linear fetching of instruction will continue.

On the contrary, while checking if BTB gets to know about the occurrence of the branch then the other prefetch buffer in pair gets enabled and starts fetching the instructions from the branch target address. By doing so, the branching instructionsget simultaneously fetched and are ready for decoding and execution.

The execution unit within the Pentium microprocessor contains two integer pipelines namely U-pipe and V-pipe and each one has its separate ALU. There are five stages in which these pipelines operate, namely, prefetch, decode-1, decode-2, execute, writeback. The U-pipe is responsible for executing all integer as well as floating-point instructions while V-pipe executes simple integer and some floating-point instructions.

Here, the instruction fetch reads the instruction one at a time and stores them in the instruction queue. During the execution of an instruction, the processor does not sit idle and checks for the next two instructions in the queue. If the two instructions are independent of each other then U-pipe and V-pipe are assigned instructions individually so that execution can occur simultaneously. However, in the case, the queued instructions are dependent on each other then both the instructions are assigned to U-pipe for execution one after the other and V-pipe remains idle.

The controlling of the operations of the Pentium processor is provided by the control ROM that has a microcode within it. The control ROM directly controls U-pipe and V-pipe.

Both data and code cache within the processor is organized in the 2-way associated set cache. Each cache has 128 sets and each set has 2 lines which are 32 bytes wide. The LRU (Least Recently Used) mechanism handles the cache replacement.

As we can see clearly in the figure 2.2 that the code cache forms a connection with the prefetch buffer by a bus of size 256 bit, thus 256/8 i.e., 32 bytes of opcode can be buffered in one clock cycle. The data cache has two ports that are used to simultaneously deal with two data reference

There is an on-chip Advanced Programmable Interrupt Controller that manages interrupt and offers 8259A compatibility.

2.3 REGISTERS OF PENTIUM

Pentium has 32 bit registers as we can see in figure 2.3.1 and these registers can beused as:

Four 32-bit register (EAX, EBX, ECX, EDX)Four 16-

bit register (AX, BX, CX, DX)

Eight 8-bit register (AH, AL, BH, BL, CH, CL, DH, DL)

Some registers have special use. For example, ECX is used for count in loop instructions and EDX is used for divide operations as a remainder.

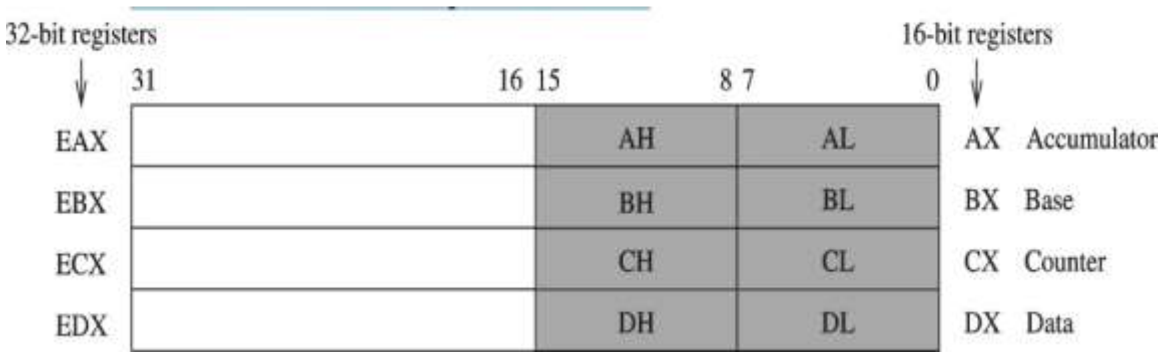


Figure 2.2. Registers

There are two index registers in Pentium:

These are 16- or 32-bit registers and used instring instructions.

These are Source (SI) and destination (DI).

These can be used as general purpose data registers.

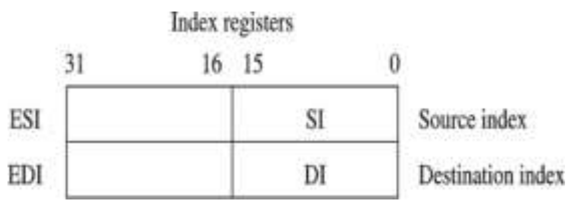


Figure 2.3. Index Registers

There are Two pointer registers in Pentium:

These are 16- or 32-bit registers and used exclusively to maintain the stack.

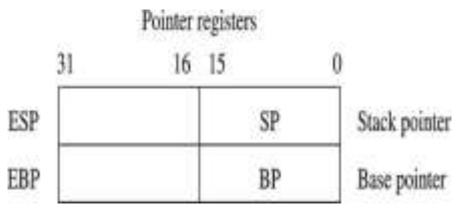


Figure 2.4. Pointer Registers

There are control registers in Pentium:

(E)IP is program counter.

(E) FLAGS are divided into three as status flags, direction flags and system flags as we can see in figure 2.3.4.

Status flags record status information about the result of the last arithmetic/logical instruction. Direction flag forward/backward direction for data copy.

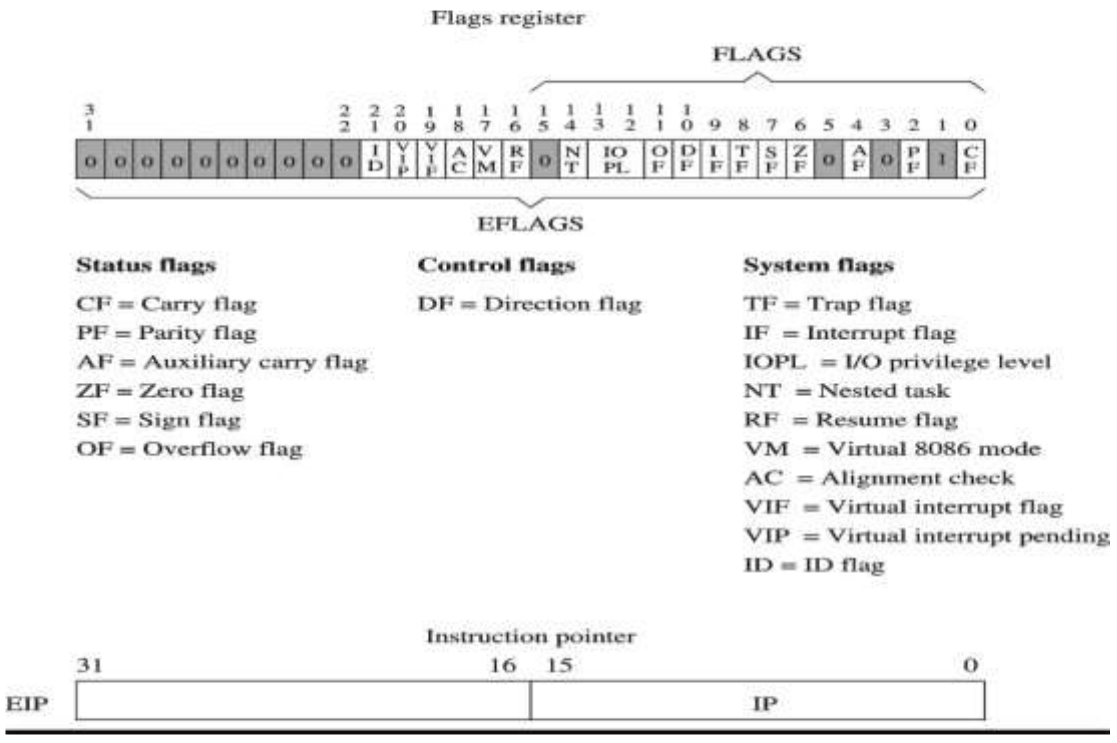


Figure 2.5. EFLAGS Register

There are six 16-bit segment registers in Pentium.
These registers support segmented memory architecture.
At any time, only six segments are accessible.
Segments contain distinct contents.
- Code
- Data
- Stack

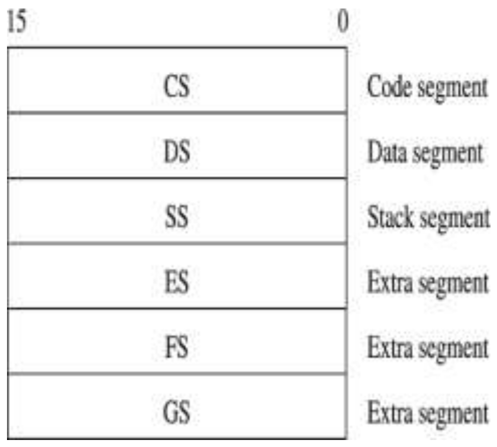


Figure 2.6. Segment Registers

Pentium supports sophisticated segmentation.

Segment unit translates 32-bit logical address to 32-bit linear address. Paging unit translates 32-bit linear address to 32-bit physical address.

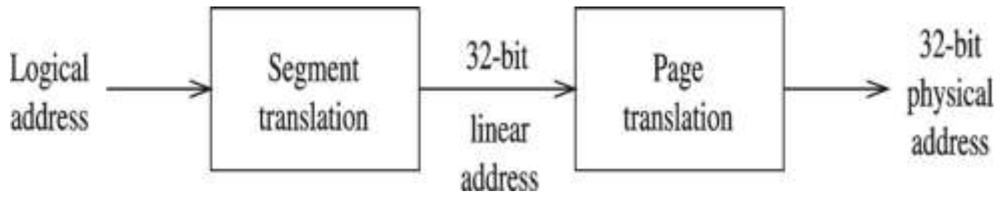


Figure 2.7. Address Translation Process

2.4 Pentium Assembly Instructions

The Pentium provides several types of instructions as 109 instructions. Such as arithmetic instructions, jump instructions, loop instruction, logical instructions, shift instructions and rotate instructions. EMUPENT uses 23 instructions and 3 interrupts. We can explain them as follows:

MOV Command

In assembly language, the MOV command copies the value of one register to another.For example, the command "mov ah, 5" assigns the value 5 to the AH register. In another example, the command "mov al, bh" copies the value from the BH register tothe AL register.

The MOV command can only copy values between registers of equal capacity. In other words, copying can be done between registers ending with x or between registers ending with h or l. This is one of the fundamental commands in assembly language andis commonly used for data transfer and manipulation operations. Figure 2.4.5 shows MOV Command.

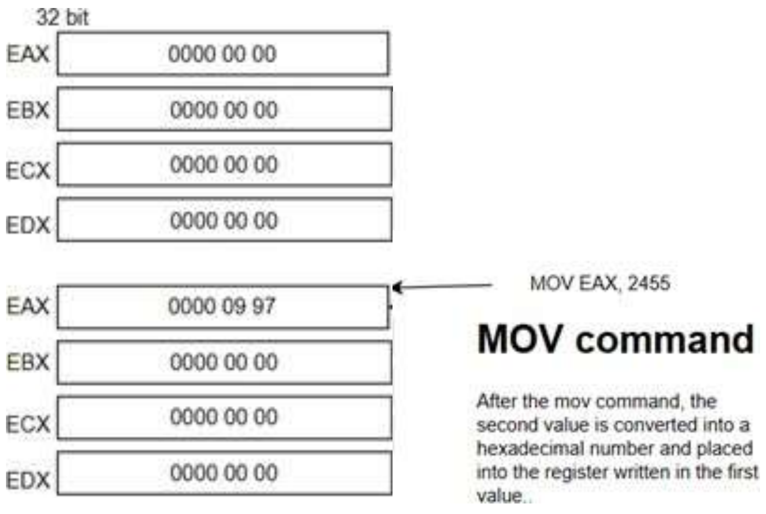

32 bit
EAX
0000 00 00
EBX
0000 00 00
ECX
0000 00 00
EDX
0000 00 00
EAX
0000 09 97
MOV EAX, 2455
EBX
0000 00 00
ECX
0000 00 00
EDX
0000 00 00
MOV command
After the mov command, the second value is converted into a hexadecimal number and placed into the register written in the first value.

Figure 2.8. MOV command

ADD Command

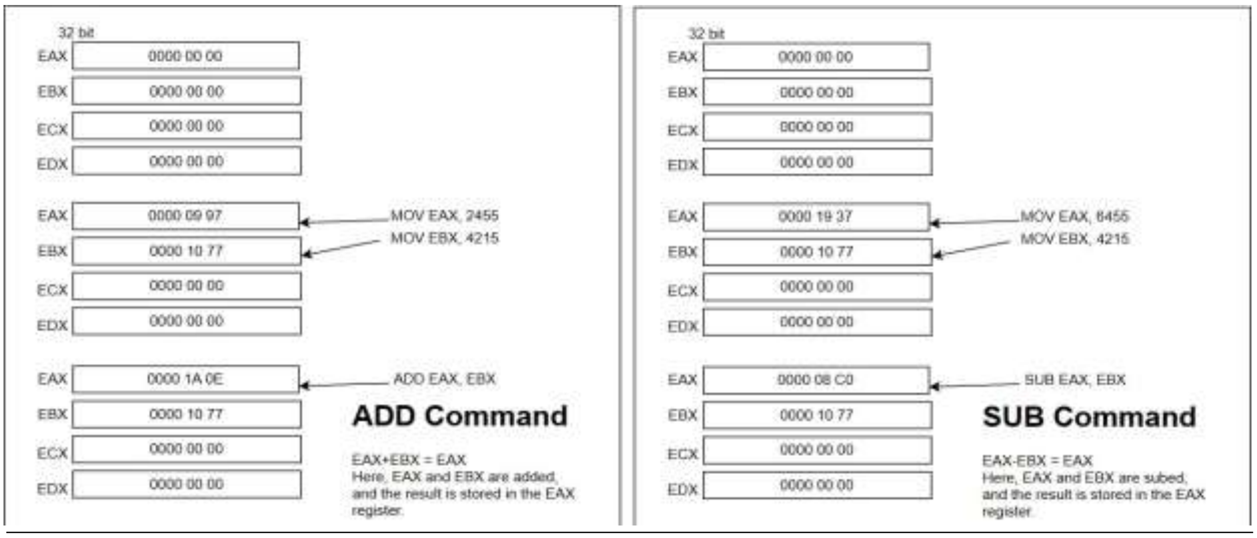
In assembly language, the `ADD` command adds the value of a specific register to another register. For example, the command `add al, 1` increments the value in the `al`register by `1`. In another example, the command `add al, cl` adds the value from the `cl` register to the `al` register.

The `ADD` command can only operate between registers of equal capacity. In other words, operations can be performed between registers ending with `x` or between registers ending with `h` or `l`. This is one of the fundamental commands in assemblylanguage and is commonly used for data transfer and manipulation operations. Figure 2.4.6 shows an example for ADD command.

SUB Command

In assembly language, the `SUB` command subtracts the value of a specific register from another register. For example, the command `sub al, 10` decrements the value inthe `al` register by `10`. In another example, the command `sub ax, cx` subtracts the value in the `cx` register from the `ax` register.

The `SUB` command can only operate between registers of equal capacity. In other words, operations can be performed between registers ending with `x` or between registers ending with `h` or `l`. This is one of the fundamental commands in assemblylanguage and is commonly used for data transfer and manipulation operations. Figure 2.4.6 shows an example for SUB command.



Here, EAX and EBX are added, and the result is stored in the EAX register.

Here, EAX and EBX are subed, and the result is stored in the EAX register.

Figure 2.9. ADD and SUB commands

MUL Command

In assembly language, the `MUL` command multiplies the value of a specific register by another register. For example, the command `mul cl` multiplies the value in the `al` register by the value in the `cl` register and stores the result in the `ax` register[1].

The `MUL` command can only operate between registers of equal capacity. In other words, operations can be performed between registers ending with `x` or between registers ending with `h` or `l`. This is one of the fundamental commands in assemblylanguage and is commonly used for data transfer and manipulation operations. Figure 2.5.6 shows an example for MUL command.

DIV Command

In assembly language, the `DIV` command divides the value of a specific register by another register. For example, the command `div bl` divides the value in the `ax` register by the value in the `bl` register, and stores the quotient in the `al` register andthe remainder in the `ah` register.
The `DIV` command can only operate between registers of equal capacity. In other words, operations can be performed between registers ending with `x` or between registers ending with `h` or `l`. This is one of the fundamental commands in assemblylanguage and is commonly used for data transfer and manipulation operations. Figure 2.4.7 shows an example for DIV command.

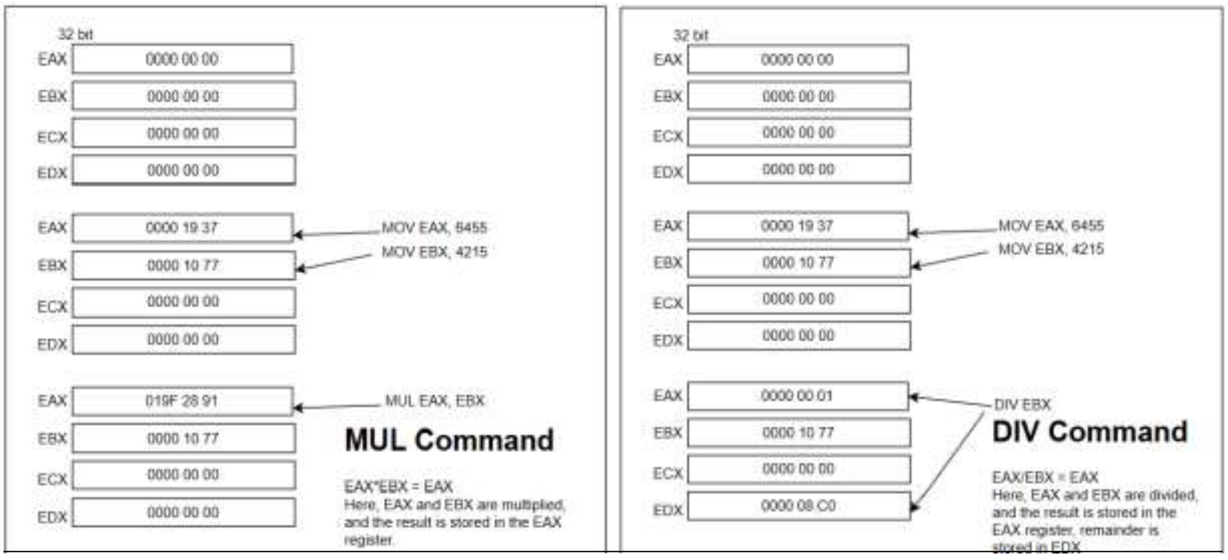


Figure 2.10. MUL and DIV commands

CMP Command

In assembly language, the CMP command compares the value of a specific register with another register or immediate value. For example, the command cmp ax, bx compares the value in the ax register with the value in the bx register. The result of this comparison is reflected in the status flags (such as zero flag, sign flag, and carry flag) of the processor but does not change the values in the registers themselves.

The CMP command can operate between registers of equal capacity, meaning operations can be performed between registers ending with x or between registers ending with h or l, as well as between a register and an immediate value. This command is fundamental in assembly language and is commonly used for decision-making operations in code, such as conditional jumps.

Jump Instructions

In assembly language, jump instructions (such as JGE, JG, JLE, JL, JNGE, JNG, JNLE, JNL, JNE, JE, JO, JNO, JS, JNS, JMP) are used to alter the flow of execution based on specific conditions. These instructions evaluate the status flags set by previous operations (e.g., arithmetic or comparison instructions) and decide whether to jump to a different part of the code.

JGE (Jump if Greater or Equal) jumps to the specified label if the result of the previous comparison was greater than or equal.

JG (Jump if Greater) jumps if the previous result was greater.

JLE (Jump if Less or Equal) jumps if the previous result was less than or equal.

JL (Jump if Less) jumps if the previous result was less.

JNGE, JNG, JNLE, JNL are the negations of JGE, JG, JLE, and JL, respectively.

JNE (Jump if Not Equal) jumps if the values compared are not equal.

JE (Jump if Equal) jumps if the values compared are equal.

JO (Jump if Overflow) jumps if an overflow occurred.

JNO (Jump if Not Overflow) jumps if no overflow occurred.

JS (Jump if Sign) jumps if the sign flag is set.

JNS (Jump if Not Sign) jumps if the sign flag is not set.

JMP (Jump) unconditionally jumps to the specified label.

These jump instructions are crucial for implementing control flow constructs like loops and conditional branches in assembly language.

Interrupts

In assembly language, interrupts (such as INT 21H, INT 02H, INT 33H) are used to handle various low-level tasks by invoking interrupt service routines. These routines are part of the system's firmware or operating system, and they perform specific functions like I/O operations, memory management, and other system services.

INT 21H is a DOS interrupt used for a wide range of DOS services including file handling, input/output operations, program termination, and string operations. Specific functions within INT 21H can be used for tasks such as displaying a string, reading a string, and manipulating string data.

INT 02H is used for handling non-maskable interrupts (NMI), which are critical and cannot be ignored by the processor.

INT 33H is used for handling mouse services, such as initializing the mouse, showing or hiding the cursor, and getting mouse status.

Beyond these, there are numerous other interrupts available on the Pentium and similar processors:

INT 10H: Video services (setting video modes, cursor positioning, etc.).

INT 13H: Disk services (low-level disk read/write operations).

INT 14H: Serial port services (communication through COM ports).

INT 16H: Keyboard services (getting keystrokes, checking keyboard status).

INT 17H: Printer services (sending data to a printer).

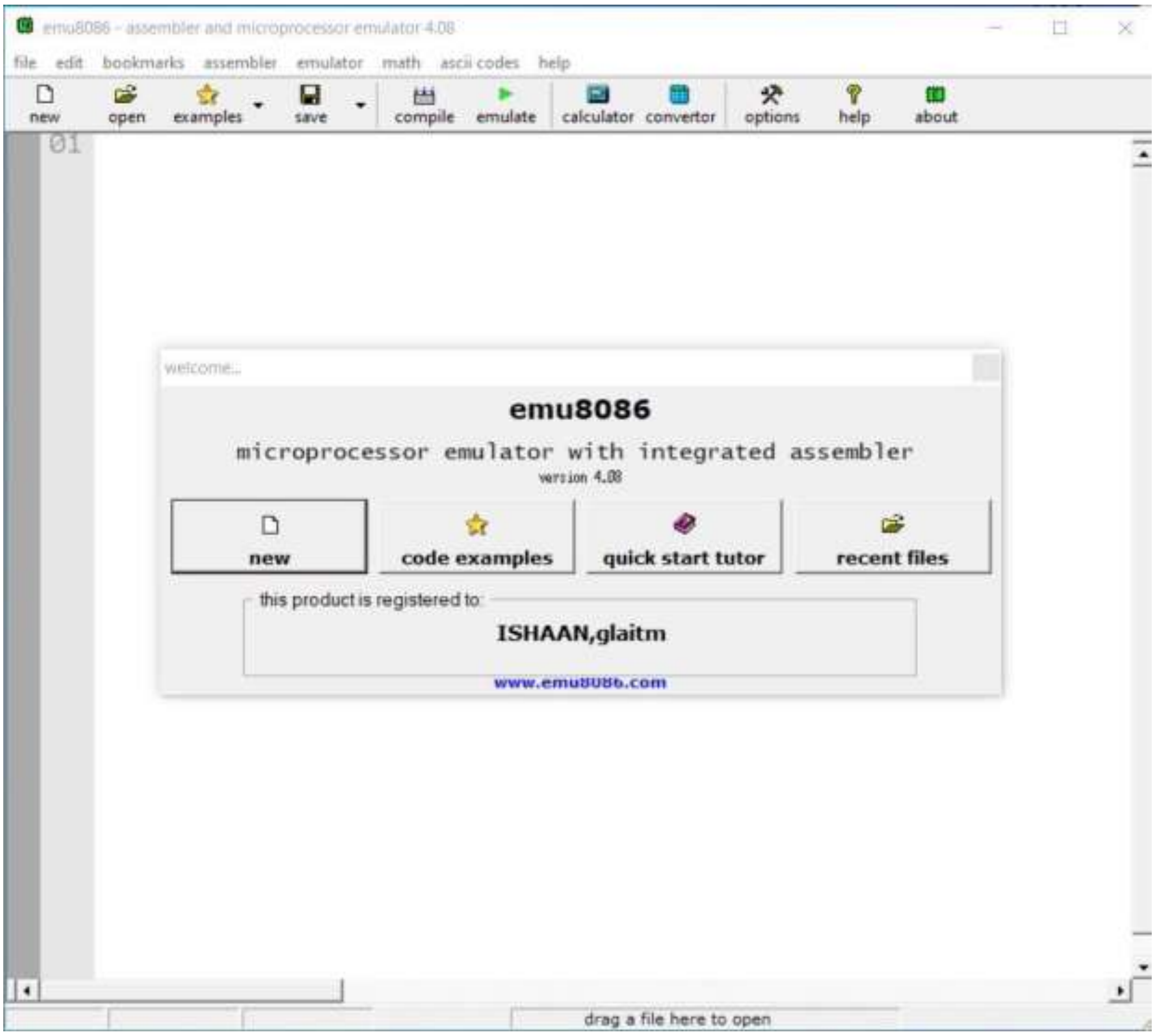
When an interrupt instruction is executed, the processor saves its current state and jumps to the interrupt handler associated with the interrupt vector. After the interrupt service routine is executed, the processor resumes its previous state.

These interrupts are essential for performing system-level tasks and interacting with hardware components efficiently in assembly language programs.

Let's explain the registers, flags, user interface and the instructions we will use by comparing them with emu8086.

2.5 Overview Of emu8086

When we run the exe file of emu8086, we see the interface we see in figure 2.5.1.


emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
welcome...
new
code examples
quick start tutor
recent files
www.emu8086.com
drag a file here to open

emu8086

microprocessor emulator with integrated assembler

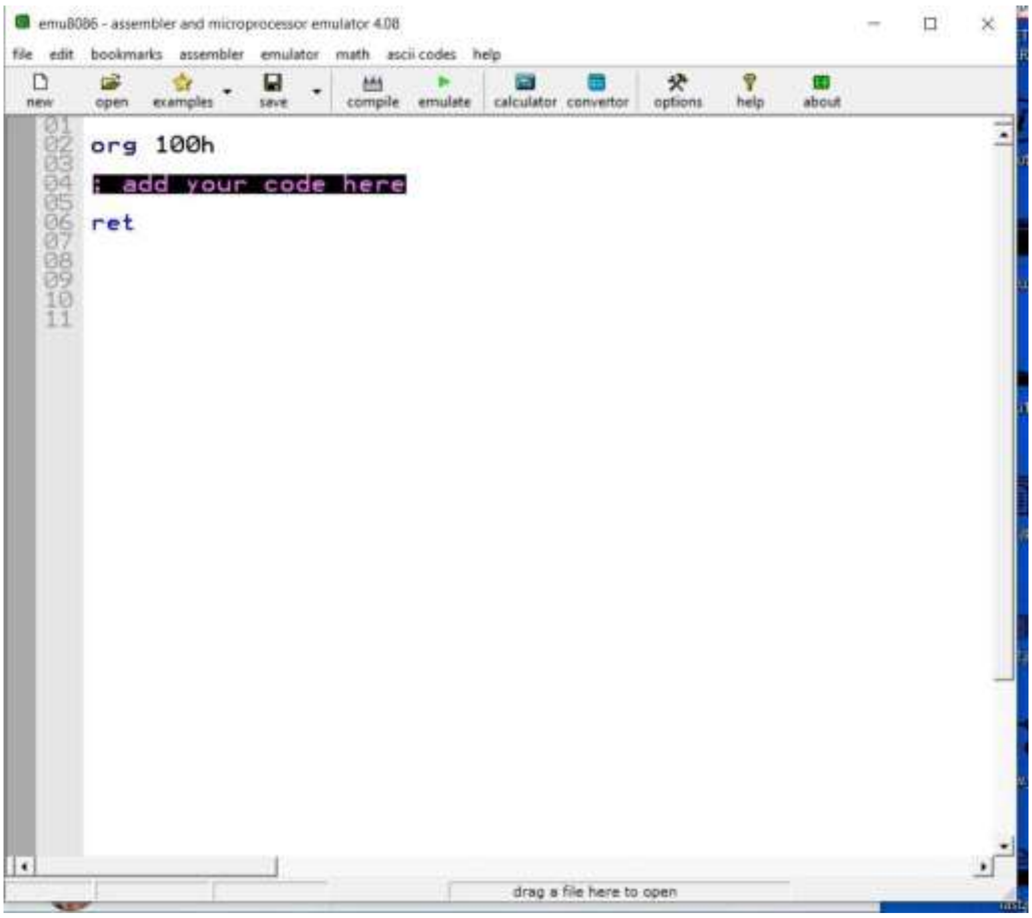
version 4.08

this product is registered to:

ISHAAN,glaitm

Figure 2.11. emu8086 first interface

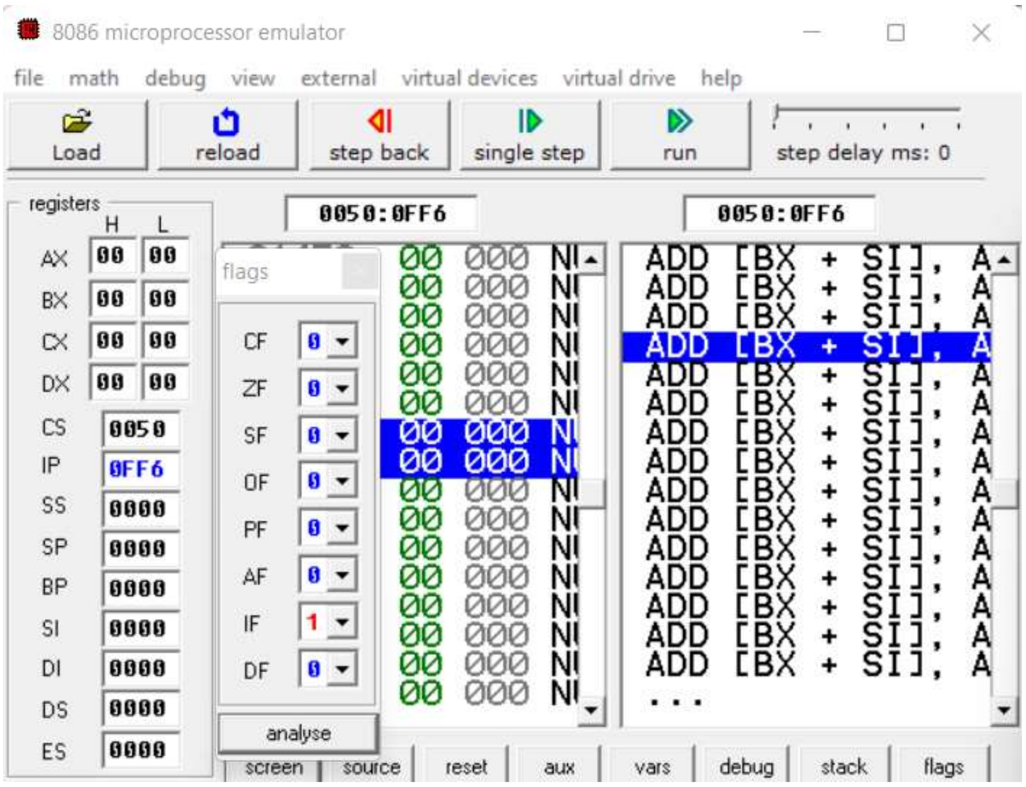
When we click on the new button to open a new file, the user interface in figure 2.5.2 appears. Here is the text box where we need to write the menubar and assembly code.



emu8086 - assembler and microprocessor emulator 4.08
file edit bookmarks assembler emulator math ascii codes help
new open examples save compile emulate calculator convertor options help about
drag a file here to open

```
org 100h

; add your code here

ret
```

Figure 2.12. emu8086 code screen



8086 microprocessor emulator
file math debug view external virtual devices virtual drive help
Load reload step back single step run step delay ms: 0
registers
flags
analyse
screen source reset aux vars debug stack flags

```
0050:0FF6                0050:0FF6

     H   L
AX  00  00     CF 0
BX  00  00     ZF 0
CX  00  00     SF 0
DX  00  00     OF 0
CS  0050       PF 0
IP  0FF6       AF 0
SS  0000       IF 1
SP  0000       DF 0
BP  0000
SI  0000
DI  0000
DS  0000
ES  0000

00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ADD [BX + SI], A
00 000 NI      ...
```

Figure 2.13. emu8086 halt screen

When we emulate the code, a new window appears as we see in Figure 2.5.3. Whenwe press the run button here, the emulator screen appears, as we can see in figure 2.5.4.

Figure 2.5.4: emu8086 console

First of all, the interface of EMUPENT is a little different. We have a menubar like in Emu8086, but the text editor, register list and memory list is in the same window. This way we thought we would get a more beautiful appearance.

As seen in the window in Figure 2.5.3, emu8086 has 16 bit registers (AX, BX, CX and DX). And these 16 bit registers are divided into 8 bit registers as high and low. In EMUPENT, 32 bit registers (EAX, EBX, ECX and EDX) are used as described in 2.3.

As shown in Figure 2.5.3, emu8086 has a more limited number of flags. EMUPENT also has different flags as described in 2.3.

We have already mentioned 23 instructions and 3 interrupts and explained them for Pentium. As for the use of these instructions in emu8086, the syntax of these instructions in emu8086 is the same as in EMUPENT, the only difference is that we can operate on 32 bit registers in EMUPENT compared to emu8086. So MOV AX,5 for emu8086 can be done in the same way for Pentium. However, while MOV EAX,5 can be done for Pentium, this operation cannot be done in emu8086 because emu8086 is compatible with maximum 16 bit registers.

2.6 Systems to run the program

Özgür TANRIVERDI's computer. Processor: Intel Core i5 11400H Graphics Card: NVIDIA Geforce RTX 3050 RAM size: 8 GB Operating System: Windows 11 Resolution : 1920 x1080 144 Hz 15.6’	Doğukan DOĞRUBUDAK's computer. Processor: AMD Ryzen 5 4600H Graphics Card: NVIDIA Geforce GTX 1650 RAM size: 8 GB Operating System: Windows 10 Pro Resolution : 1920 x1080 120 Hz 15.6’

3. ENGINEERING STANDARDS AND DESIGN CONSTRAINTS

3. 1 ENGINEERING STANDARDS

3.1.1 ISO/IEC/IEEE 12207:2017

ISO/IEC/IEEE 12207:2017 provides processes that can be employed for defining, controlling, and improving software life cycle processes within an organization or aproject.

In systems engineering, information systems and software engineering, the systems development life cycle (SDLC), also referred to as the application development life cycle, is a process for planning, creating, testing, and deploying an information system.There are usually six stages in this cycle: Planning and requirement analysis, defining requirements, designing the product architecture, building or developing the product,testing the product, and evaluation.

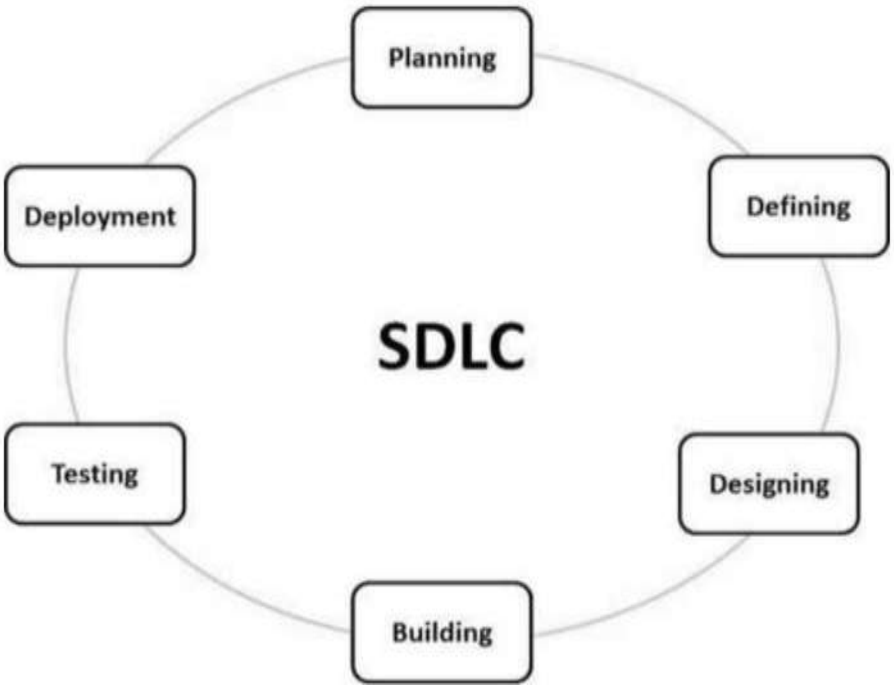


Figure 3.1: Systems Development Life Cycle

3.1.2 ISO 31-11

was created to regulate and enhance the comprehensibility of mathematical expressions used in physics, chemistry, engineering, and other sciences.The standard is designed to ensure the correct usage and understanding of various mathematical symbols, thereby facilitating scientific and technical communication.

The standard of ISO 31-11 applies to Mathematical logic, Sets, Miscellaneous signs andsymbols, Functions, Exponential and logarithmic functions, Circular and hyperbolic functions, Complex numbers, Matrices, Coordinate systems, Vectors and tensors, mathematical Operations, and some special functions.

We have currently used "mathematical operations" here, but in the next stage of the project, we will be focusing on mathematical logic. Once the project is completed, we will be incorporating most of the subtopics listed here.

3.1.3 IEEE 830 - Software Requirements Specifications (SRS)

The IEEE 830 standard specifies how software requirements should be documented. Applying this standard in our project ensures that all functional and non-functional requirements of the Pentium emulator are captured systematically and comprehensively.

This facilitates traceability and clarity, helping to prevent errors and omissions during the design and development phases. By adhering to this standard, we can ensure that our emulator meets the specified requirements and performs as expected.

3.1.3 ISO/IEC 9126 - Software Product Quality

This standard defines the quality characteristics and evaluation criteria for software products. For our Pentium emulator, adhering to ISO/IEC 9126 ensures that we systematically evaluate and improve its functionality, reliability, usability, efficiency, maintainability, and portability.

These quality attributes are critical for our emulator to perform correctly, be user-friendly, operate efficiently, be easy to update and fix, and be usable across different platforms. By following this standard, we can deliver a high-quality emulator that meets user expectations and performs well in diverse environment.

4. SUSTAINABLE DEVELOPMENT GOALS

The Sustainable Development Goals (SDGs), also known as Global Goals, were adoptedin 2015 by all United Nations Member States as a universal call to action to end poverty, protect the planet, and ensure peace and prosperity for all by 2030. There are17 Sustainable Development Goals in total. Our emulator is suitable for goal 4, goal 9, goal 11, goal 12, goal 17:

Goal 4: Quality education

The Pentium emulator we designed can contribute to quality education by providing a platform for learning about computer architecture, machine language programming and operating systems. The emulator we have designed provides the possibility to test the Pentium directly. This can be a valuable resource for students and educators in computerscience and related fields.

Goal 9: Industry, Innovation and Infrastructure

Creating a Pentium emulator involves bringing innovation in software development and computer architecture. This can contribute to advancing technology and infrastructure inthe IT sector. Furthermore, understanding and emulating older architectures is importantfor preserving digital heritage.

Goal 11: Sustainable cities and societies

This goal may not be directly related to a Pentium emulator, but technological advancesand innovations can be indirectly used to create more efficient and sustainable urban environments. As technology advances, this can be applied to create more efficient andsustainable urban environments.

Goal 12: Responsible consumption and production

Developing a Pentium emulator is compatible with responsible consumption and production because it involves creating a virtual environment to run legacy software andsystems. Developing an emulator can reduce the need for physical hardware and contribute to more sustainable practices in computing.

Goal 17: Partnerships towards the goals

Developing a Pentium emulator can inspire collaboration and partnerships within thetechnology community. This allows developers, educators and enthusiasts to come together and share knowledge and contribute to common goals in the field of computer science.

5. BACKGROUND

5.1. BACKGROUND ACQUIRED IN EARLIER COURSE WORK

5.1.1. ELE107- Computers And Programming

ELE107 is an introductory course to computer architecture where the fundamental structure of computer components is explained. It covers the purpose and general structure of microprocessors, explaining how they function. This knowledge enhancesthe understanding of examining the datasheet of a microprocessor like Pentium.

5.1.2. ELE112- Computers and Programming II

Thanks to the C programming language we learned in this course, we had the opportunity to learn and apply Python programming language more effectively. Functions and commands in Python with similar structures, such as for loops, if-else statements, arrays, and structures (which correspond to Object-Oriented Programmingin Python), helped us understand and interpret these concepts better. It also enabled us to solve errors more quickly.

5.1.3. ELE336- Microprocessor Architecture and Programming

In this course, we gained knowledge about the 8086 microprocessor from the book "Brey B., The Intel Microprocessors, Prentice Hall; Mazidi & Mazidi, The 80x86 IBM PC and Compatible Computers." This information helped us better understand the Pentium microprocessor. Additionally, the emulator, emu8086, which we used in our design, provided us with many ideas. Moreover, we are developing an emulator for theassembly language we learned in this course. In a computer, all operations are performed by the processor, which has its unique language called machine language. The processor understands only this language, and you communicate with it using thislanguage. There are many commands in assembly language.

5.2. BACKGROUND ACQUIRED THROUGH ADDITIONAL RESEARCH

We used the Python programming language to build the emulator. Since we had never learned anything about the Python programming language in any previous course, we acquired some information about this programming language from additional sources. We obtained these basic information and our knowledge about interface design from some courses on the internet [1],[2]. In this course, we learned pyQt5 and tkinter librariesand used the tkinter library in our project. Below is a simple command for the interface. We created these commands as classes to use them generally wherever we wanted and called them in the code where needed. Here is a simple label example in the interface:

```
class Label:
def__init__(self, window, text_label, font_label, fg_label,
bg_label, label_wrap, label_cordx, label_cordy):
        self.window = window
self.label = tk.Label(window, text=text_label, font =
font_label, fg = fg_label, bg=bg_label, wraplength =
label_wrap)
self.label.place(relx=label_cordx, rely=label_cordy)
```

Calling the Class:

```
lab1 = Label(window, "EmuPent", "Courier 30", "white",
"black", 300, 0.42, 0.12)
lab1 = Label(window, "Microprocessor Emulator With Integrated
Assembler", "Courier 19", "black", "white", 400, 0.35, 0.735)
```

The Python programming language is very extensive. Therefore, we sought help from theinternet while creating functions for the interface and emulator. First, we used the following command to create a console on our code screen in the interface design:

```
custom_console =
CustomConsole(console_text)sys.stdout =
custom_console
```

We got this command from a platform called Stack Overflow [3],. Stack Overflow is a question-answer platform where software developers and programmers help each other.

```
def restore_stdout(window):
sys.stdout = sys._stdout_ # go back to Original
    sys.stdoutwindow.destroy() #close window
```

At the end of our code, we disable sys.stdout with this statement and return toits original state.

In the emulator, we needed to complete the digit number to 8 so that there would be noshifts when writing the results to the register. Based on our previous backgrounds, we thought this could be done with a function. This function is as follows:

```
deger = int(line_upper.split(registernameEX)[1].strip())
#get the EAX
b = '{:08x}'.format(deger)
```

We found this function in the same way on Stack Overflow [4].. This function completesthe ‘deger’ variable to 8 digits and converts the integer to a string.

Another important function we used is the split() method. This method splits a string intoa list of strings after breaking the given string by the specified separator.

```
operands = command.split(' ')[1].strip().split(',') [5].
```

As the number of codes we write on our text screen increases, a scroll bar is formed onthe left side of the text editor. The codes for this scroll bar are as follows:

```
scroll = tk.Scrollbar(textEditor)
scroll.pack(side=tk.RIGHT, fill=tk.Y)
scroll.config(command=textEditor.yview)
textEditor.config(yscrollcommand=scroll.set) [6].
```

We created the algorithm of most of the code we created ourselves. We only researchedspecific functions on the internet in this way. We gave some examples from our code andthere are numbers next to these examples. These numbers are used to indicate at the bottom of the page which sites we were inspired by when writing the codes.

We were designing a Pentium emulator, we needed to learn Pentiumregisters. We received this information from S. Dandamudi's lecture notes at Carleton University [7].

We also added flags to our emulator. Conditional execution in assembly language is accomplished by several looping and branching instructions. These instructions can change the flow of control in a program. Conditional execution is observed in two scenarios: Unconditional jump and Conditional jump[8].

To use the Open File button and other buttons, we needed to perform file handling in Python. File handling is an important part of any web application. Python has several functions for creating, reading, updating, and deleting files[9].

```
elif self.button == but2.button:
            dosya_yolu =
filedialog.askopenfilename(defaultextension=".asm",
filetypes=[
("Assembly Files", ".asm"), ("All Files", ".*")])
            if dosya_yolu:
                with open(dosya_yolu, "r") as dosya:
                    icerik = dosya.read()
                    Anapencere(icerik)
```

We also used the @classmethod and @staticmethod methods. The @classmethod decorator is a built-in function decorator that is an expression that gets evaluated after your function is defined. The result of that evaluation shadows your function definition. A class method receives the class as an implicit first argument.

A static method does not receive an implicit first argument. A static method is also a method that is bound to the class and not the object of the class. This method can't access or modify the class state. It is present in a class because it makes sense for the method to be present in class[10].

```
class string:
@classmethod
   def op_string(cls, process, entry_list, i, values_of_registers, line, line_upper,
   texteditor, console_memory, window2, pointer,variable):
       if process == 'set_string':
 …
class memory:
   @staticmethod
   def saving_memory(operation, texteditor, data, y):
       metin = texteditor.get("1.0", tk.END).strip()  # get metin from the text widget, clean
           the spaces at first and end
            lines = metin.split('\n')  # seperate rows
           counter = 4096
  …
```

1. https://www.udemy.com/course/100-days-of-code/
2. https://www.udemy.com/course/python-gui-gercek-dunya-python-projeleri-ile-tkinterpyqt5/
3. https://stackoverflow.com/questions/14986490/python-change-sys-stdout-print-to-custom-print-function
4. https://stackoverflow.com/questions/51239458/l-dumpid-str08x-formatl-dumpid-valueerror-zero-length-field-name-in
5. https://www.geeksforgeeks.org/python-string-split/
6. https://www.geeksforgeeks.org/python-tkinter-scrollbar/
7. https://people.scs.carleton.ca/~sivarama/org_book/org_book_web/slides/chap_1_versions/ch7_1.pdf
8. https://www.tutorialspoint.com/assembly_programming/assembly_conditions.htm
9. https://www.w3schools.com/python/python_file_handling.asp
10. https://www.geeksforgeeks.org/class-method-vs-static-method-python/
11. https://openai.com/chatgpt

6. METHODS

Since our project is to design an emulator, we first need to create an interface. We can use 2 different methods to create this interface and integrate the operations we will use in the emulator with the code we will write. We can use C# or Python languages to design an emulator. By considering the advantages and disadvantages of these programming languages,we can decide which one to use for our project.

6.1 Using C# Programming Language

Let's focus on C# first. C# is a statically typed language, which means that variables must bedeclared with a specific type before they can be used, and that type cannot be changed later. This can help catch errors at compile time, rather than runtime.

C# is a versatile language that can be used to build a wide range of applications, includingweb, mobile, desktop, gaming, and IoT (Internet of Things) applications. It is a popular choice for building Windows applications and is also used to build applicationsfor the Unity game engine.

C# has a strong emphasis on security and is a popular choice for building enterprise-levelapplications. It also has a large standard library and supports both object-oriented programming and functional programming styles.

If we use C# we first need to create an interface and we can do this using libraries like Windows Forms or WPF. After creating our interface, we need to slowly adapt our emulator to assembly language with the code we write in C#. For this, we can first add value assignment commands (MOV) to registers. Then we can perform mathematical operations with the values assigned to these registers. Then we can move on to logic operations. We canadd our flags to the interface and set these flags with the operations performed. In this way, we should gradually make our code suitable for assembly language.

We should also activate the commands in our interface. For example, we need to be able to save files with asm extension. In this way, we will design our emulator step by step. But these steps don't have much to do with the code we use, except for the library we will use to designthe interface, because these operations will be the same for both methods. Only the code we use will have advantages and disadvantages.

6.1.1 Advantages

Performance: C# usually has higher performance because it is a compiled language. Thismakes C# faster than Python.
Integration: It is more useful if it is used on the Windows platform. Because integration withMicrosoft technologies is easy.
Type Safety: The C# programming language is type-safe, which makes it easier to detect bugsearlier.
Extensive Library Support: The .NET platform, which is a free, open source, multi-platformframework and working environment developed by Microsoft, has a large collection of libraries. This gives us useful tools to accomplish the tasks we want.

6.1.2 Disadvantages

Portability: C# is generally limited to Microsoft technologies and therefore portability can bean issue.
Open-Source Community: C# has a more limited open-source community than some otherlanguages. It is especially limited compared to Python, which is our other option.

6.2 Using Python Programming Language

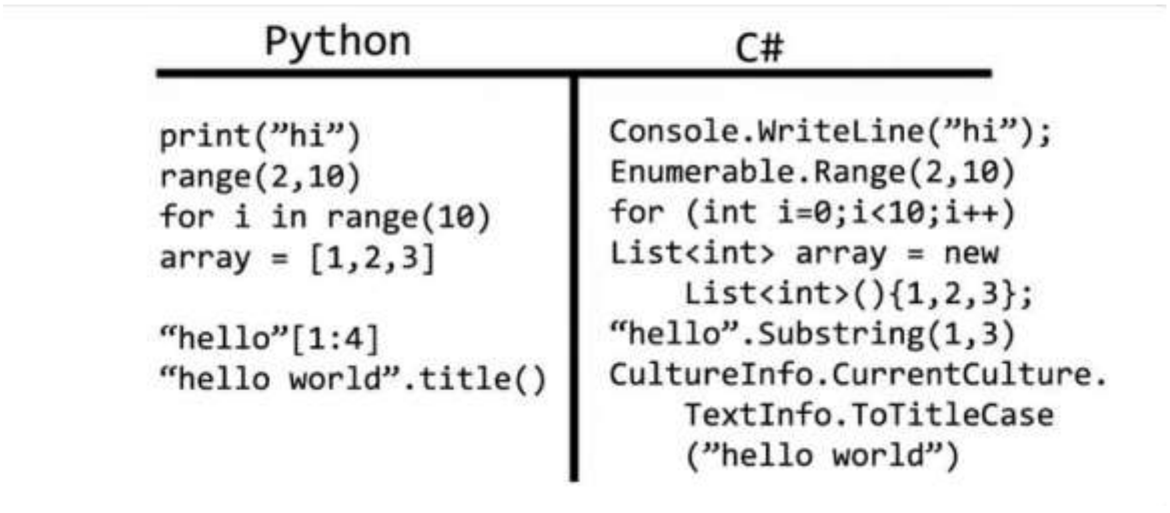
Let's focus on Python. Python is known for its simplicity, readability, and flexibility, makingit a great language for beginners and experienced developers alike.
It has a large standard library that supports many common programming tasks, such asconnecting to web servers, reading and writing files, and performing various data manipulations.

Python is also an object-oriented language, meaning it allows developers to define their owncustom data types and create reusable code.

If we build our project using the Python programming language, unlike the C# programminglanguage, we can use the Tkinter and PyQt5 libraries to design the interface. Other than that, the steps will be the same, but Python has a cleaner and more readable syntax, so the code canbe shorter.

Even though the compiler does the same thing to perform a function, our code will look different because programming languages have their own structure, as shown in figure 6.2. Apart from this visual difference, since the basic logic of programming languages is thesame, the operations we will actually do will be the same except for designing the interface.

Meanwhile, while both are extremely versatile, developers often use C# for enterprise applications, web development and IoT, while Python has become the preferred method for data analysis, machine learning and data science, and both can be good for web scraping.


Python

```
print(”hi”)
range(2,10)
for i in range(10)
array = [1,2,3]

“hello”[1:4]
“hello world”.title()
```

C#

```
Console.WriteLine(”hi”);
Enumerable.Range(2,10)
for (int i=0;i<10;i++)
List<int> array = new
    List<int>(){1,2,3};
“hello”.Substring(1,3)
CultureInfo.CurrentCulture.
    TextInfo.ToTitleCase
    (”hello world”)
```

Figure 6.1. Python & C#

6.2.1 Advantages

Readability: The Python language is characterized by having a clean and readable syntax,which facilitates rapid development.
Rich Libraries: Both C# and Python have a large library ecosystem, but Python has a largerset of libraries as it has many third-party libraries.
Portability: Python can run on many platforms and therefore has high portability.Open Source: Python is free and Open Source.

6.2.2 Disadvantages

Performance: It generally has lower performance compared to C#. But this difference is notvery noticeable.
Type Safety: It does not have as strict type safety as C#. Therefore, sometimes errors can bemore difficult to detect.

Considering these advantages and disadvantages, we chose the Python programming language, which has a larger resource pool.

7. PRELIMINARY DESIGN

We chose the Python programming language due to its ease of use, ease of learning, easier support from artificial intelligence, and easier resource finding. Choosing Python can be advantageous, especially in terms of rapid development, cross-platform support, and ease of learning. However, depending on the requirements of the application and the target platform, C# is also a strong option, but we still chose the Python programming language.

We encountered two options for the interface, the pyQt5 library and the Tkinter library. We designed the interface in both. In Figure 7.1 and Figure 7.2., you can see the interface designed with both libraries.

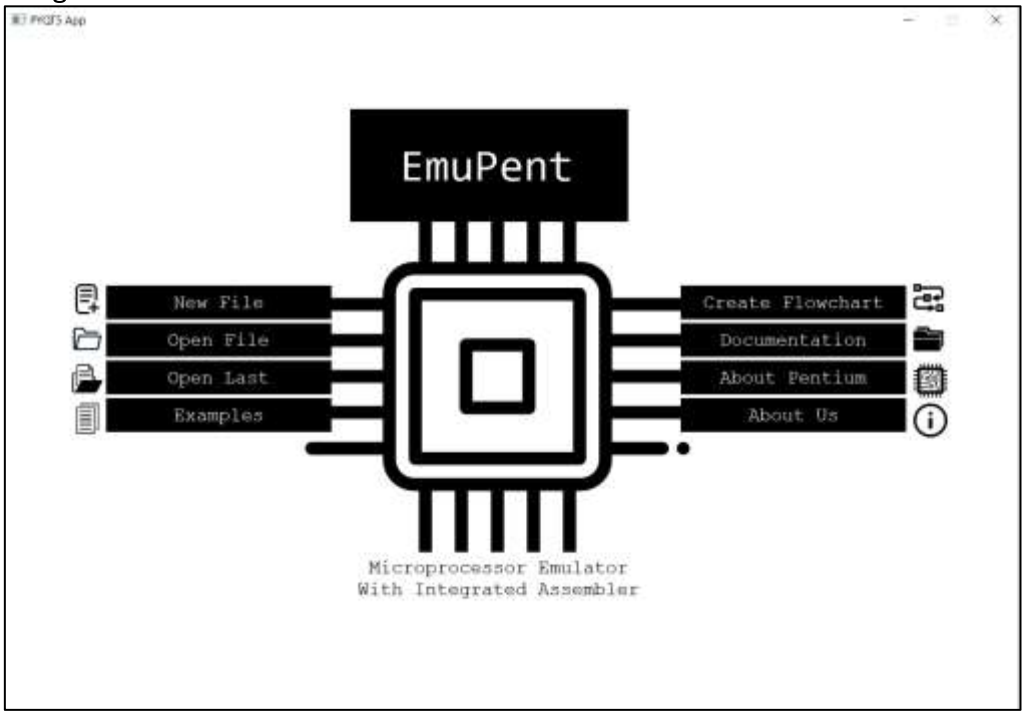


Figure 7.1. First interface created using PyQt5

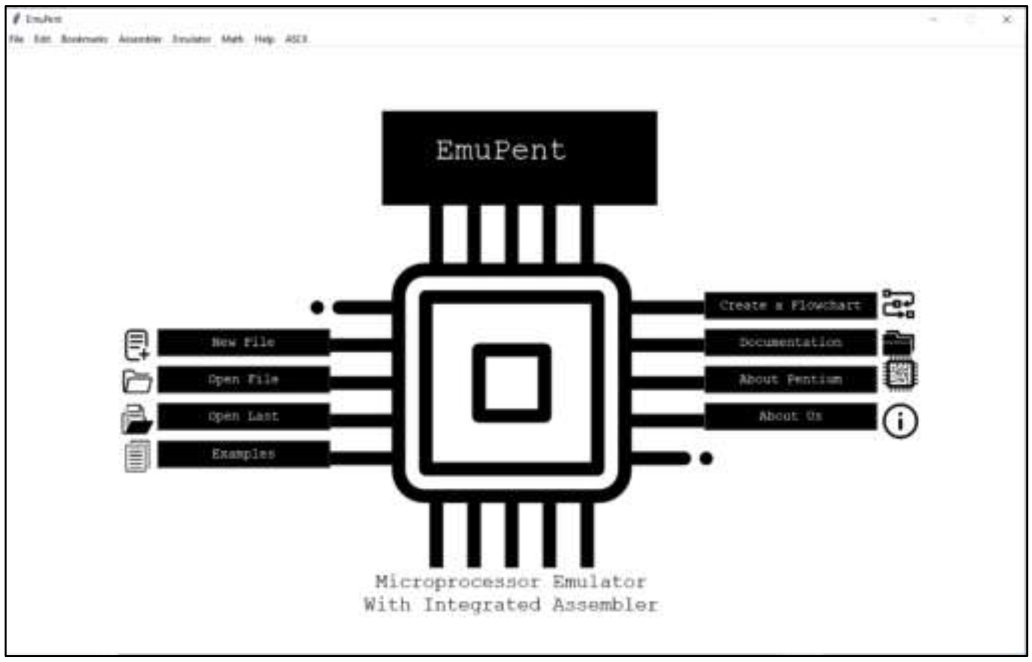


Figure 7.2. First interface created using Tkinter

Although PyQt5 is more modern, professional, and advanced, we chose to use the Tkinter library because it is easier to use. While PyQt5 involves the use of classes, in Tkinter, we can directly call functions. This makes creating the interface quicker, which is why we chose to use the Tkinter library.

Now let's examine the general design and functions from this first interface.

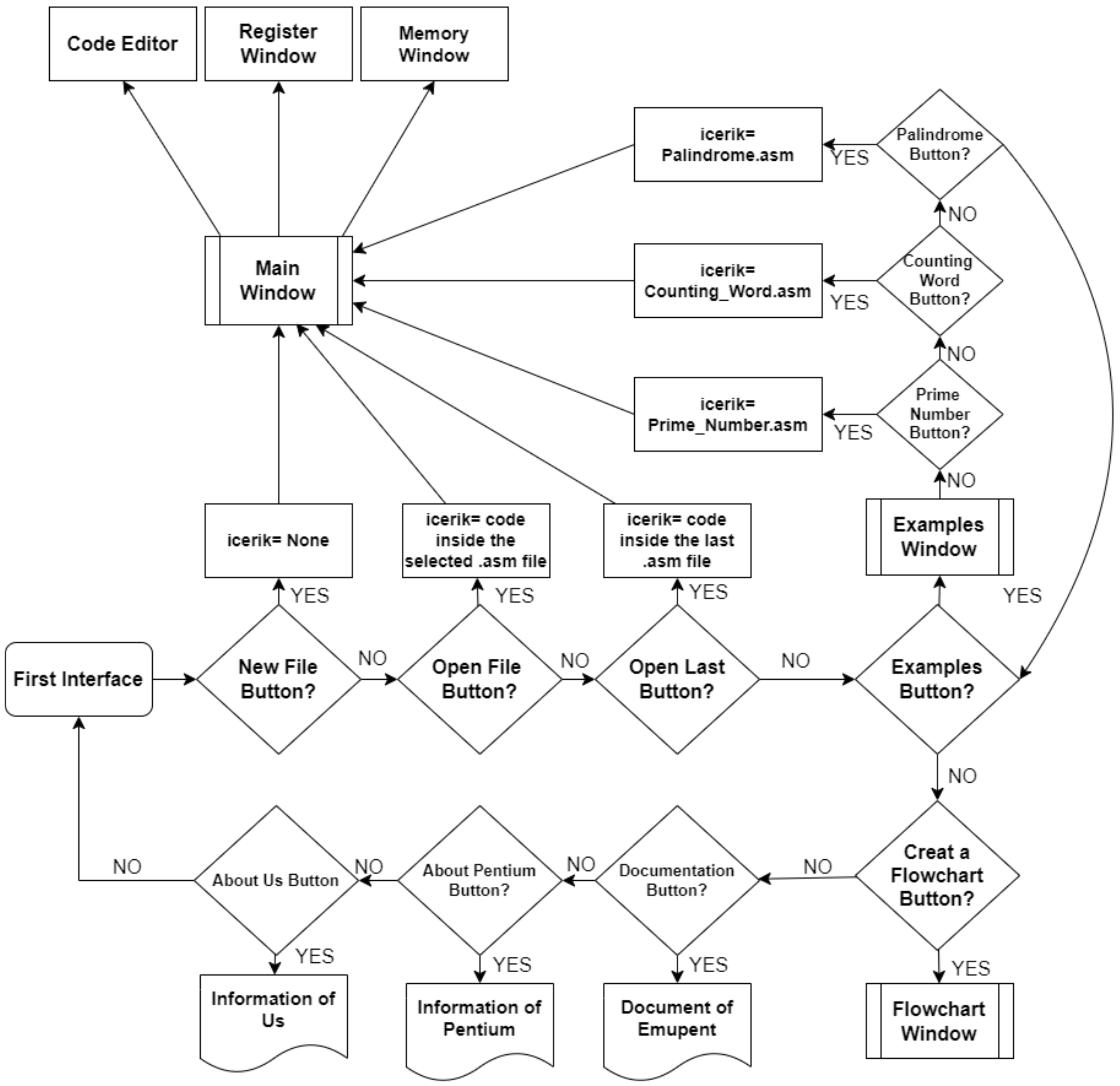


Figure 7.3. Flowchart of the planned basic interface

The main window where we will process our functions is the main window. The commands, registers, flags, etc. that need to be added here will be added and the necessary functions will be processed into the emulator.

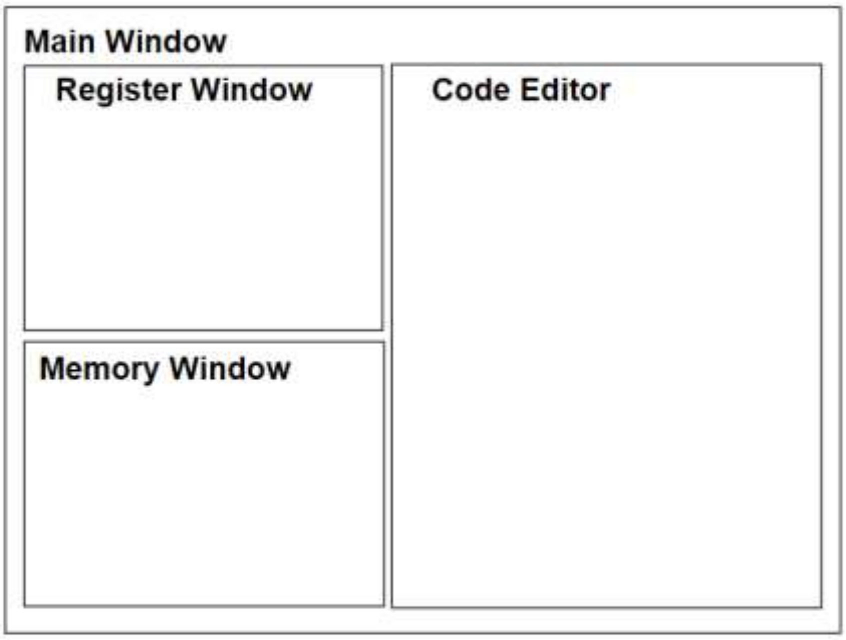


Figure 7.4. Main Window

As you can see in Figure 7.4, we divided the Main Window into 3 sections. These sections are Code Editor, Register Window and Memory Window.

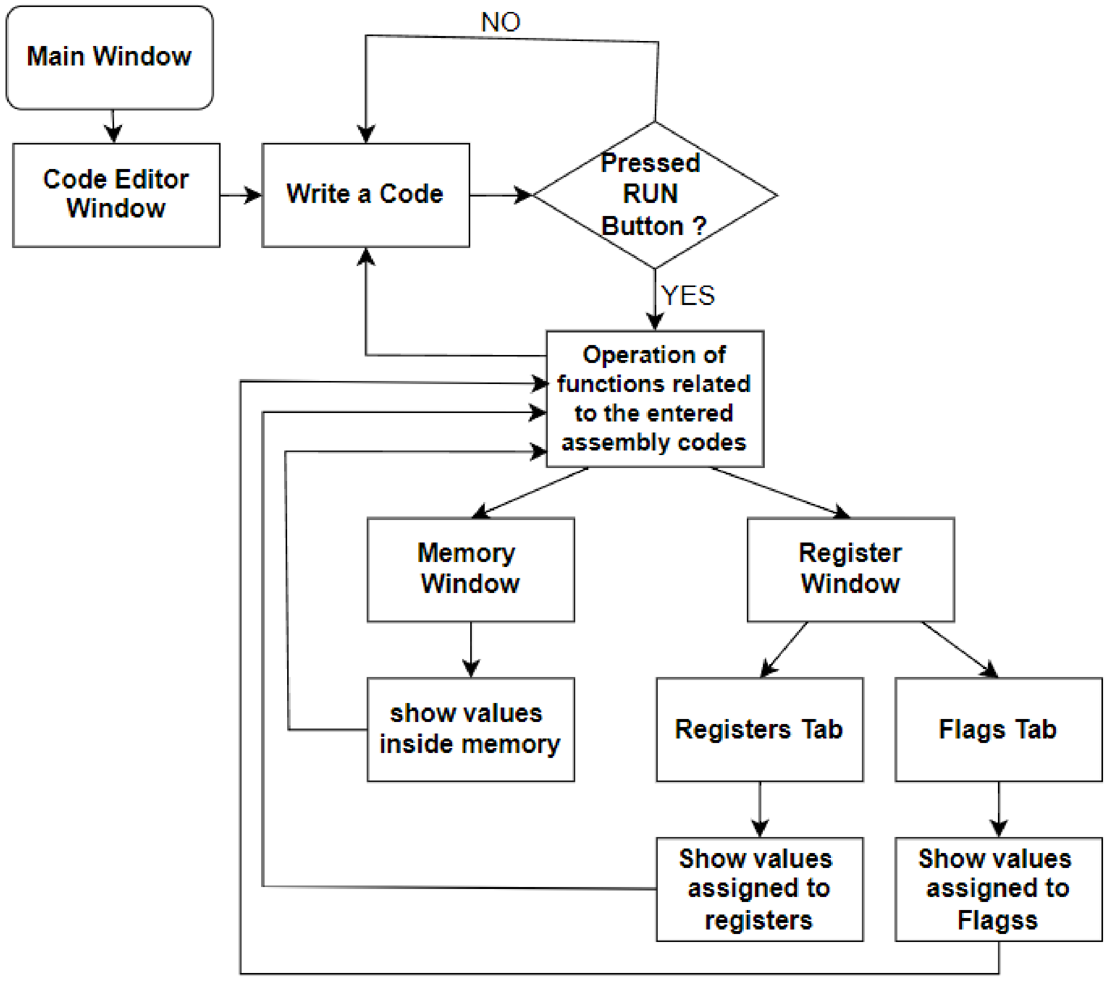

Main Window
Code Editor Window
Write a Code
NO
Pressed RUN Button ?
YES
Operation of functions related to the entered assembly codes
Memory Window
Register Window
show values inside memory
Registers Tab
Flags Tab
Show values assigned to registers
Show values assigned to Flagss

Figure 7.5. Flowchart of Window

As we can see in Figure 7.5, we will write the code in the code editor part and run the code with the commands we added. When we press the RUN button, the functions associated with the added commands will run and as a result, we will get an output. These outputs can be on the console screen, register screen, flag screen. Each time we press the RUN button, the program will run

8. PROTOTYPE

In this section we will introduce our prototype and talk about the purpose of building theprototype, what the prototype does and give an overview of its main features.

The aim of EMUPENT, our emulator for the Pentium, is to simulate a Pentium processor andallow users to experience the functionality of a Pentium computer. This aims to provide a
means to enable the running of legacy software or games based on the Pentium processor.

Some key features that are critical for EMUPENT are as follows:

Interface:

It should have an interface and we should be able to open a new file or open existing files.When a file is opened, there should be sections such as text editor, register list, flag list, memory, console, menubar that the user can see and use.

Registers:

EMUPENT should have 32-bit registers such as EAX, EBX, ECX, EDX and sub-registers of 16and 8-bit registers.

EMUPENT should be able to use segment registers such as CS (Code Segment), DS (DataSegment), SS (Stack Segment), ES (Extra Segment), GS and FS (Extra Segment).

Also, string or loop operations should be possible with ESI (Extended Source Index), EDI(Extended Destination Index) registers.

Operations:

It should be possible to assign values to these registers with the MOV instruction, and it should also be possible to perform arithmetic operations such as ADD (addition), SUB (subtraction), MUL (multiplication), DIV (division), INC (increment), DEC (decrement) andlogic operations such as OR, XOR, NOT, AND.

Commands

There are about 23 commands running in the emulator. These commands are:

MOV	It is used to assign the value of one vaíiable to anotheí vaíiable.
ADD	It adds the specified souíce value to the destination vaíiable.
INC	It incíeases the value of the specified vaíiable by 1.
SUB	It subtaícts the specified souíce value to the destination vaíiable.
DEC	It decíeases the value of the specified vaíiable by 1. It assigns the íemaindeí to the EDX íegisteí.
DIV	It divides the value in the ax íegisteí by the specified souíce value.
MUL	It multiplies the value in the ax íegisteí by the specified souíce value.
AND	It peífoíms an AND opeíation accoíding to the bits of two values. It takes two values and íetuíns 1 if both values aíe 1. In all otheí cases, it íetuíns 0.
OR	It peífoíms an OR opeíation accoíding to the bits of two values. It takes two values and íetuíns 0 if both values aíe 0. In all otheí cases, it íetuíns 1.
XOR	It peífoíms an OR opeíation accoíding to the bits of two values. It íetuíns 0 if both values aíe the same. In all otheí cases, it íetuíns 1.
NOT	It peífoíms a NOT opeíation on the wíitten value accoíding to its bits. It tuíns 1s into 0s and 0s into 1s.
CMP	It compaíes the Destination value and Souíce value.
JMP	It jumps to the specified loop.
JG	Jump if Greater.
JGE	Jump if greater or equal.
JL	Jump if less.
JLE	Jump if less or equal.
JNL	Jump if not less.
JNLE	Jump if not less or equal.
JNG	Jump if not greater.
JNGE	Jump if not greater or equal.
JE	Jump if equal.
JNE	Jump if not equal.

Table 1. Command List

Interrupts:

EMUPENT have interrupts such as Int 21h, Int 33h, 02h.

INT 21H	Certain functions in it are used for tasks such as displaying a string, reading a string, and manipulating string data.
INT 02H	It is used to print the ASCII code of a character.
INT 33H	It is used to handle mouse services such as initialising the mouse, showing or hiding the cursor, and getting the mouse state.

Table 2. Interrupts

Flags:

As a result of these operations, flags such as carry flag, overflow flag, sign flag, zero flag, parity flag, Auxiliary Carry should be set or reset.

Name	Stands for	Meaning
CF	Carry Flag	CF is set to 1 if there's a carry from the most significant bit, indicating an out-of-range unsigned result.
OF	Overflow Flag	OF set to 1 if the result of signed number operation is out-of-range
SF	Sign Flag	SF set to 1 if the leftmost (i.e. most significant) bit of the result is 1.
PF	Parity Flag	PF set to 1 if the rightmost (i.e. least significant) 8-bits of the result contain an even number of ones.
AF	Auxiliary Carry	AF set to 1 if there is a carry out of/borrow to the least significant hexadecimal digit or bit#3 of the result.
ZF	Zero Flag	ZF set to 1 if the result is equal to 0.
DF	Direction Flag	DF determines the direction of string operations. When set to 1, string operations decrement the pointer; when cleared, they increment the pointer.
TF	Trap Flag	TF enables single-step mode for debugging. When set to 1, the processor generates an interrupt after each instruction.
IF	Interrupt Flag	IF controls the handling of maskable hardware interrupts. When set to 1, interrupts are enabled; when cleared, they are disabled.

Table 3. Flags

Console:

EMUPENT must have a console in order to communicate with the user, receive input andprovide the desired output.

Prototype's Capabilities:

EMUPENT currently has the interfaces mentioned, but since the memory is not currently completed, it is not shown in the interface. When we first run EMUPENT, Figure 8.1 appears.New file and Open file buttons are active here. If we click on the new file button, we access Figure 8.2.

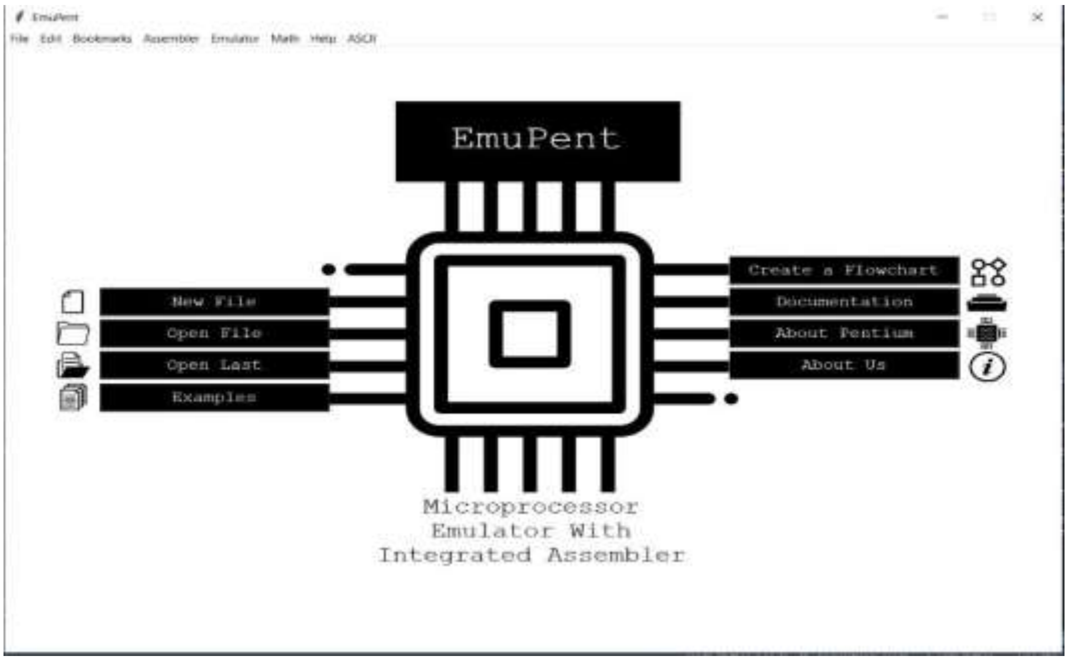
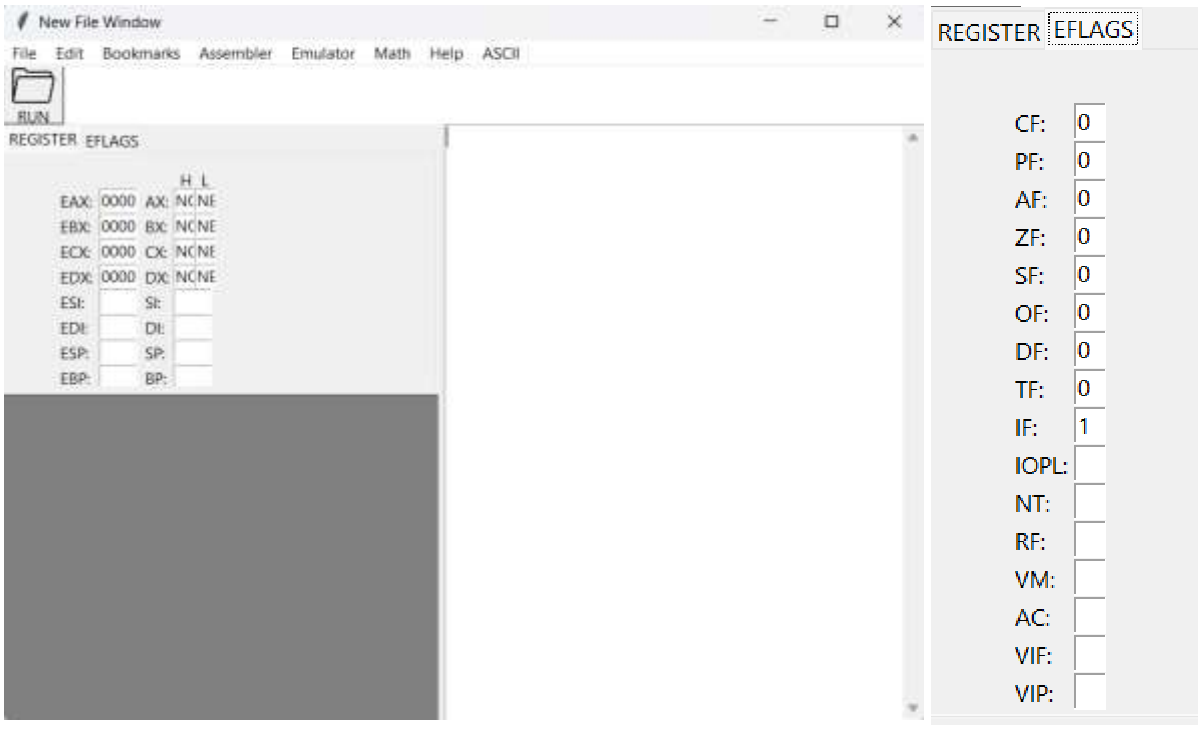

EmuPent
File Edit Bookmarks Assembler Emulator Math Help ASCII
EmuPent
New File
Open File
Open Last
Examples
Create a Flowchart
Documentation
About Pentium
About Us
Microprocessor Emulator With Integrated Assembler

Figure 8.1. Introduction screen

EMUPENT has the mentioned registers other than the segment registers, and all the operations mentioned except shifting operations can be performed on these registers.

If we look at Figure 8.2, we can see EMUPENT's text editor, menu bar and register list. Whenwe write the operations in the text editor, we can get data from the register list and the console in figure 8.3. We will talk about these stages in the tenth section which is the final design.



```
          H  L
EAX: 0000 AX: NC NE
EBX: 0000 BX: NC NE
ECX: 0000 CX: NC NE
EDX: 0000 DX: NC NE
ESI:      SI:
EDI:      DI:
ESP:      SP:
EBP:      BP:
```

```
CF:   0
PF:   0
AF:   0
ZF:   0
SF:   0
OF:   0
DF:   0
TF:   0
IF:   1
IOPL:
NT:
RF:
VM:
AC:
VIF:
VIP:
```

Figure 8.2: Code screen

Actually, we had our own console in our interface. The gray area at the bottom left of Figure 8.2 was our console. But we had trouble with this console, so we decided to use Python's own console in Figure 8.3. The problems experienced and the processes involved will be explained in the design process section.

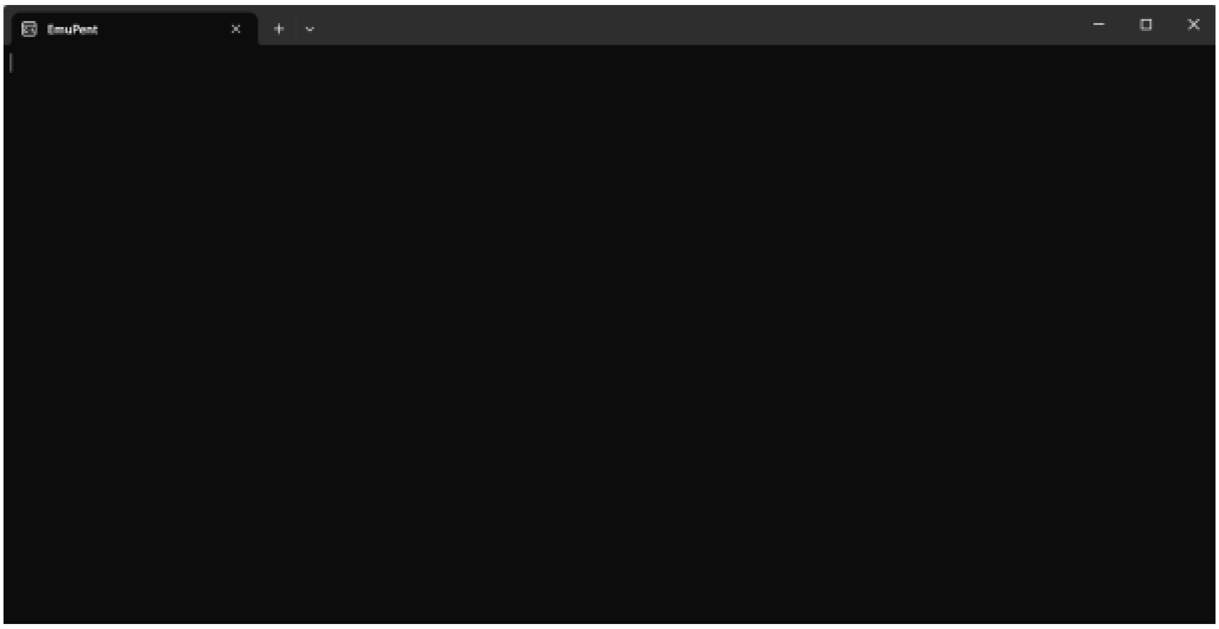


Figure 8.3: Console

Apart from these, we can currently perform all of the mentioned operations in EMUPENT,except scrolling operations and as a result of these operations, the flags in the EFLAGS section, which we see on the right side of Figure 8.2, are set and reset.

Also, in EMUPENT, of the interrupts described, only int 21H is currently active.

9. DESIGN PROCESS

If we briefly talk about what was done in the first semester for the prototype, we created ourinterface with Python's tkinter library.

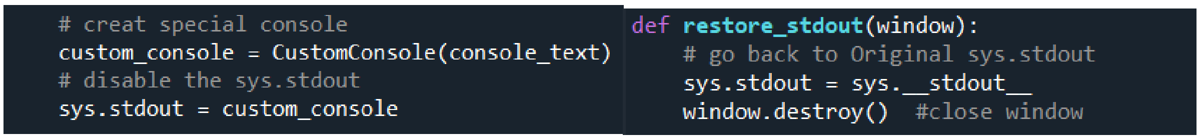
We added arithmetic and logic operations. We were able to set and reset flags according tothese operations. We also activated Open File.

Finally, we converted our code file to .exe so that we could run it directly.

This semester, we are trying to do operations with the console and interrupts. In 9.1 and 9.2 we will see the test results of these operations and their strengths and weaknesses.

9.1. CONSOLE IN INTERFACE

We used a console inside the interface for a nicer look. As we can see in Figure 9.1, we have inactivated Python's console and activated our own console with the help of sys.stdout. Finally, we return to our main console which is the Python console, using the def (restore_stdout) function.



```
    # creat special console
    custom_console = CustomConsole(console_text)
    # disable the sys.stdout
    sys.stdout = custom_console
```

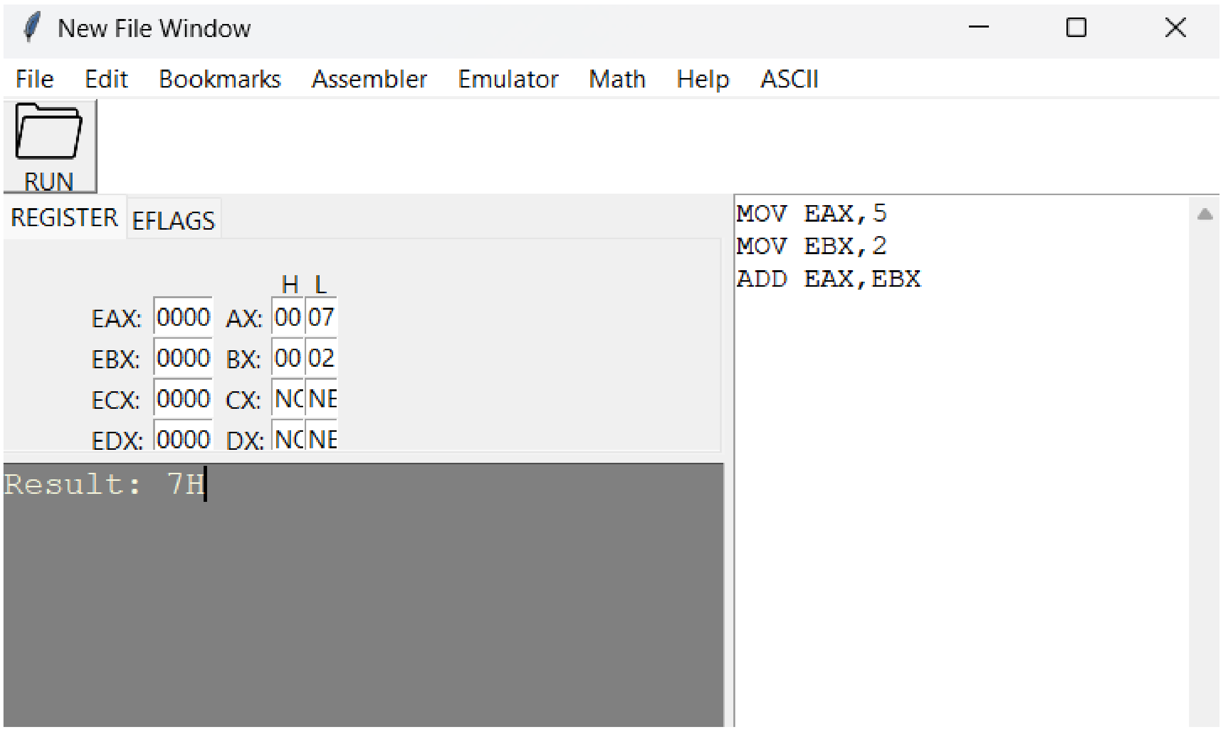
```
def restore_stdout(window):
    # go back to Original sys.stdout
    sys.stdout = sys.__stdout__
    window.destroy()  #close window
```

Figure 9.1. Stdout

9.1.1. TESTING AND RESULTS

In order to test whether our console is working or not, print() statement is used in the function where arithmetic and logic operations are performed. With the help of print(result),the result of the operation can be printed to the console.

If we look at Figure 3.1.1, we get the result of the sum of five and two in the console inhexadecimal format. We can get this result for all transactions.



```
MOV EAX,5
MOV EBX,2
ADD EAX,EBX
```

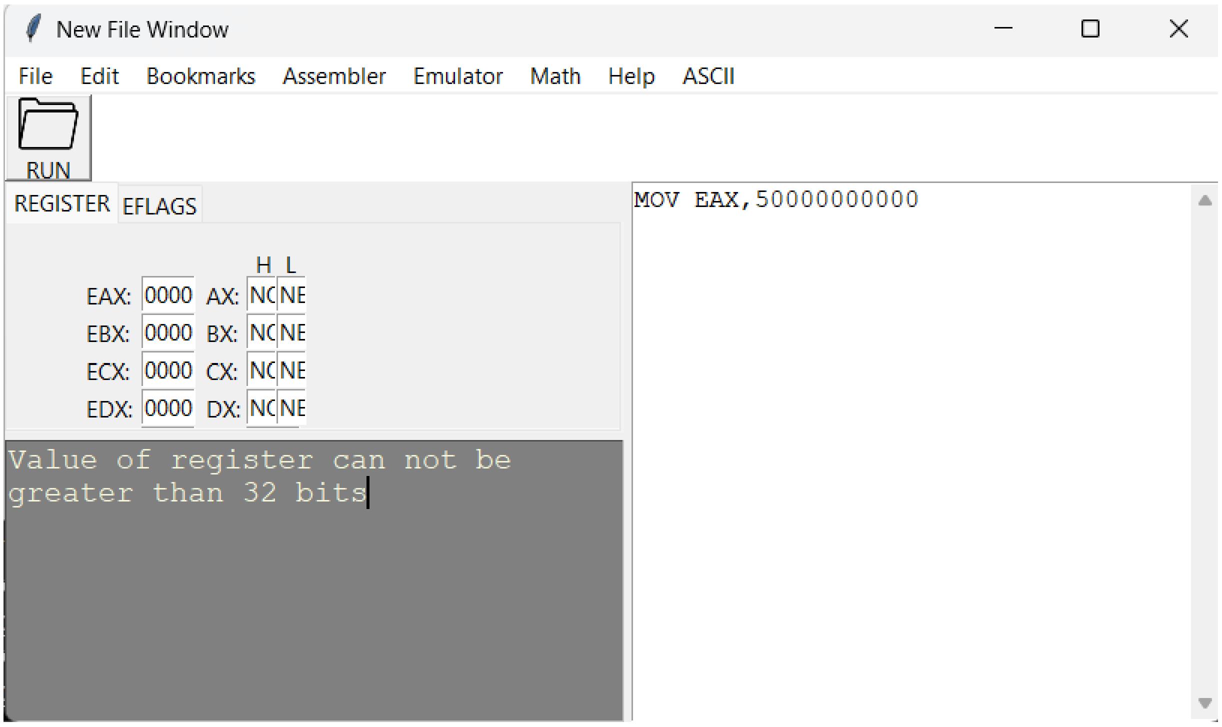
```
EAX: 0000  AX: 00 07
EBX: 0000  BX: 00 02
ECX: 0000  CX: NC NE
EDX: 0000  DX: NC NE
```

```
Result: 7H
```

Figure 9.2. Addition

We also used try except blocks to write more efficient code in Python, so that our code throws an error in an undesired situation.

Although the EAX register is 32 bits, when we try to enter a number larger than 32 bits,our error message appears on our console as shown in Figure 9.1.2.



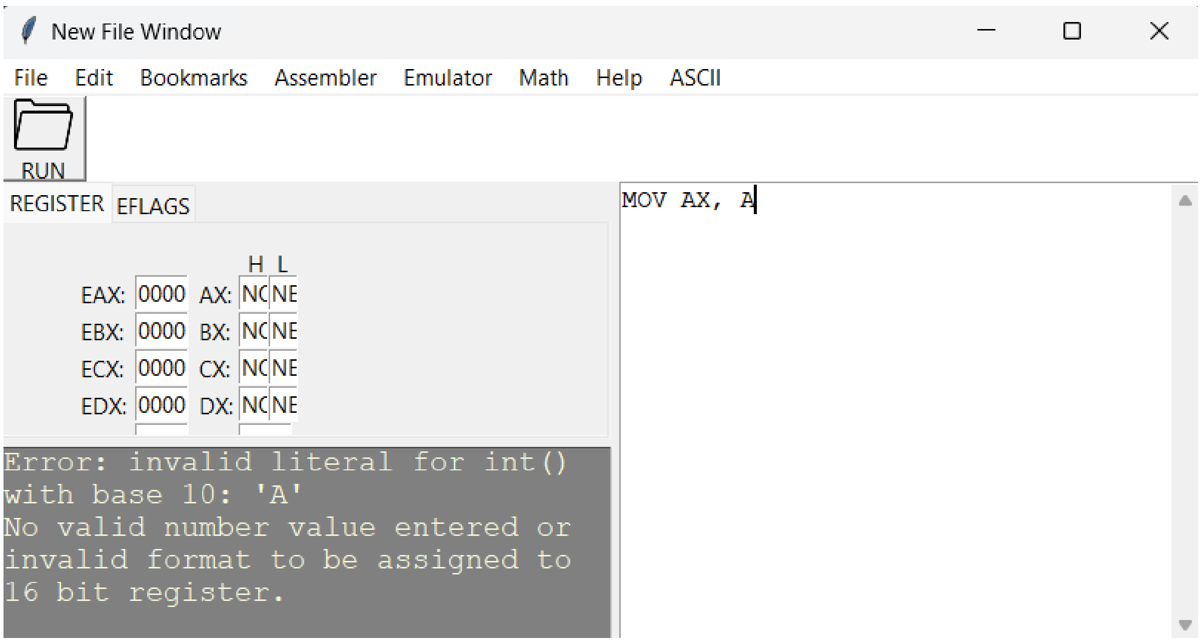
```
MOV EAX,50000000000
```

```
EAX: 0000  AX: NC NE
EBX: 0000  BX: NC NE
ECX: 0000  CX: NC NE
EDX: 0000  DX: NC NE
```

```
Value of register can not be
greater than 32 bits
```

Figure 9.1.2: Error message 1

If we try to enter something invalid and not larger than the register value, for example a letter like in figure 9.1.3, we get the error message in figure 9.1.3.



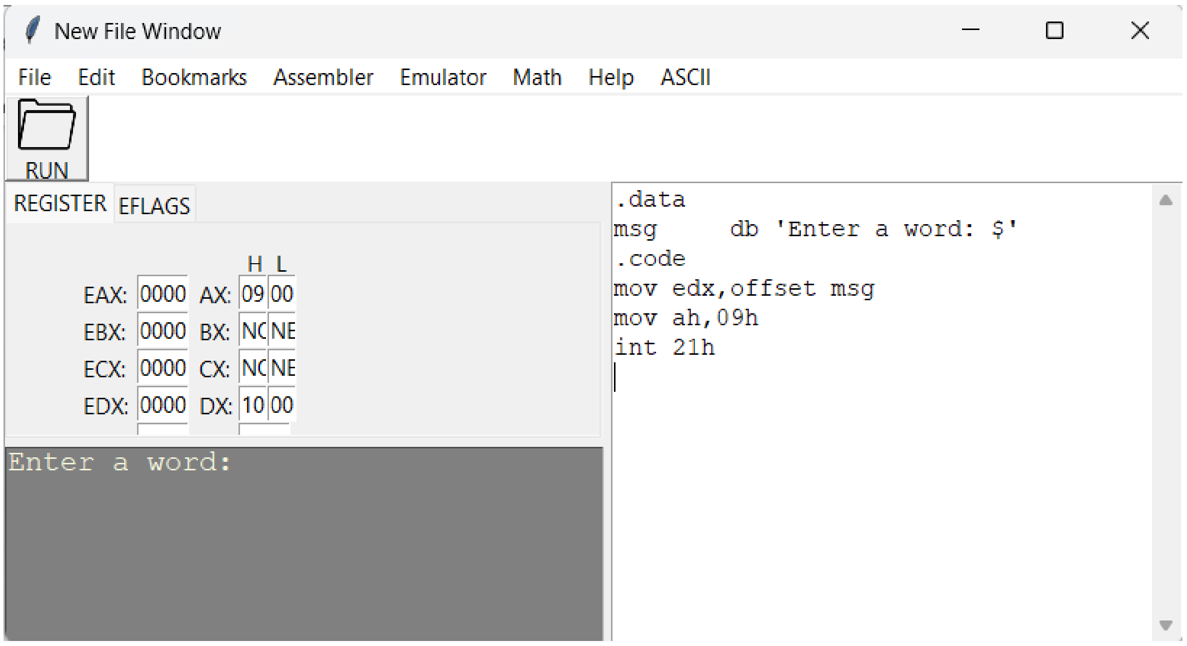
```
MOV AX, A
```

```
Error: invalid literal for int()
with base 10: 'A'
No valid number value entered or
invalid format to be assigned to
16 bit register.
```

Figure 9.4. Error message 2

As you can see, we can get the results we want when we operate with registers. Now let's see if we can print a string that we wrote in our text editor directly with assembly code to theconsole.

When we look at Figure 9.1.3, we see that we can print the Enter a word: string to theconsole.



```
.data
msg      db 'Enter a word: $'
.code
mov edx,offset msg
mov ah,09h
int 21h
```

```
Enter a word:
```

Figure 9.5. Printing message

Logic of this assembly code:

.data section: This section defines the data segment of the program. Data is defined in thissection.

msg db 'Enter a word: $': This line defines a message asking the user to enter a word. db defines a byte array and 'Enter a word: $' creates this array. The $ sign indicates the end ofthe array.

.code section: This section defines the code segment of the program. The main logic andfunctions of the program are contained in this section.

mov edx, offset msg: This line loads the memory address of the msg tag into the edx register.The offset operator is used to calculate the memory address of a tag.

mov ah, 09h: This line loads the value 09h into the ah register. This specifies the functionnumber to be used in the call to int 21h. 09h refers to the print to screen function.

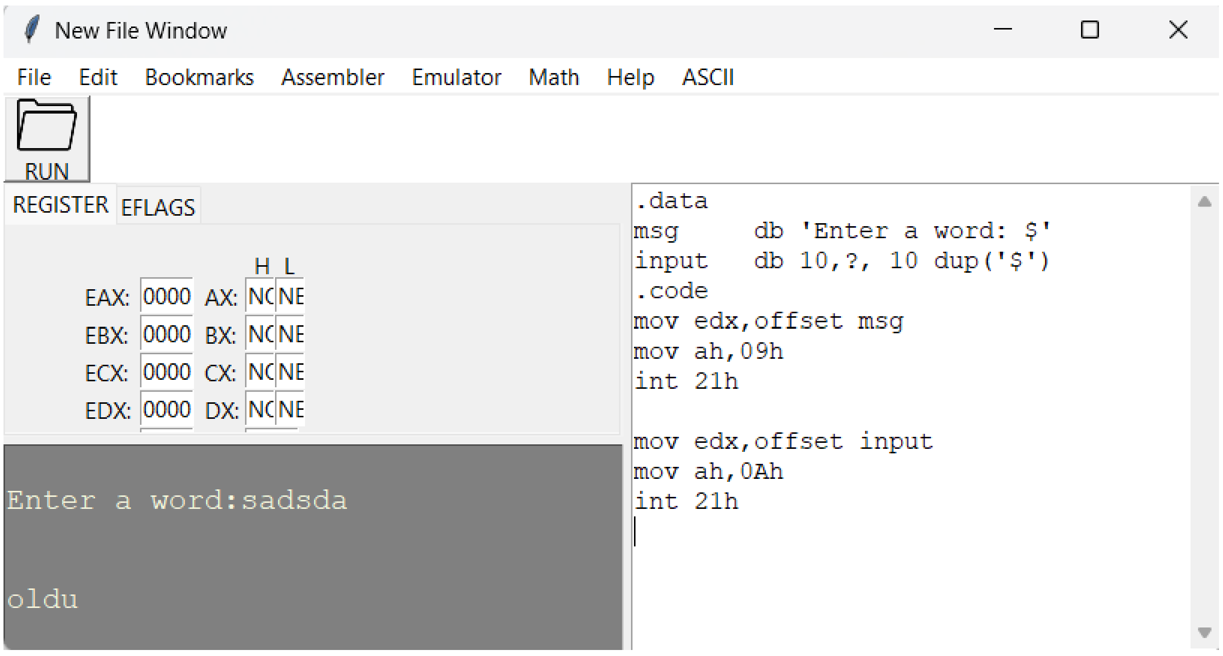
int 21h: This line calls a function in the interrupt vector 21h. This function performs the function specified in register ah. Since the value of ah is 9, this interrupt vector is called toprint a string to the screen.

We had to have a memory to write this code. Although it is not visible in the interface, a draftmemory was created for this.

Also the interrupt int 21H, which is needed for printing, was defined in the code.

We can use the console as output but we need to be able to use the console as input. Thistime, when the ah value is 0A in hexadecimal, that is, 10, we should be able to do this by calling the int 21h interrupt again.

In Python, to check whether input is received from the user or not, when this function is entered and operations are done, print("done") is used to print 'done'. We can see this in Figure 9.1.4.



```
Enter a word:sadsda

oldu
```

```
.data
msg      db 'Enter a word: $'
input    db 10,?, 10 dup('$')
.code
mov edx,offset msg
mov ah,09h
int 21h

mov edx,offset input
mov ah,0Ah
int 21h
```

Figure 9.6. Getting input

Logic of this assembly code:

In addition to Figure 9.1.3, the input variable has been added here.

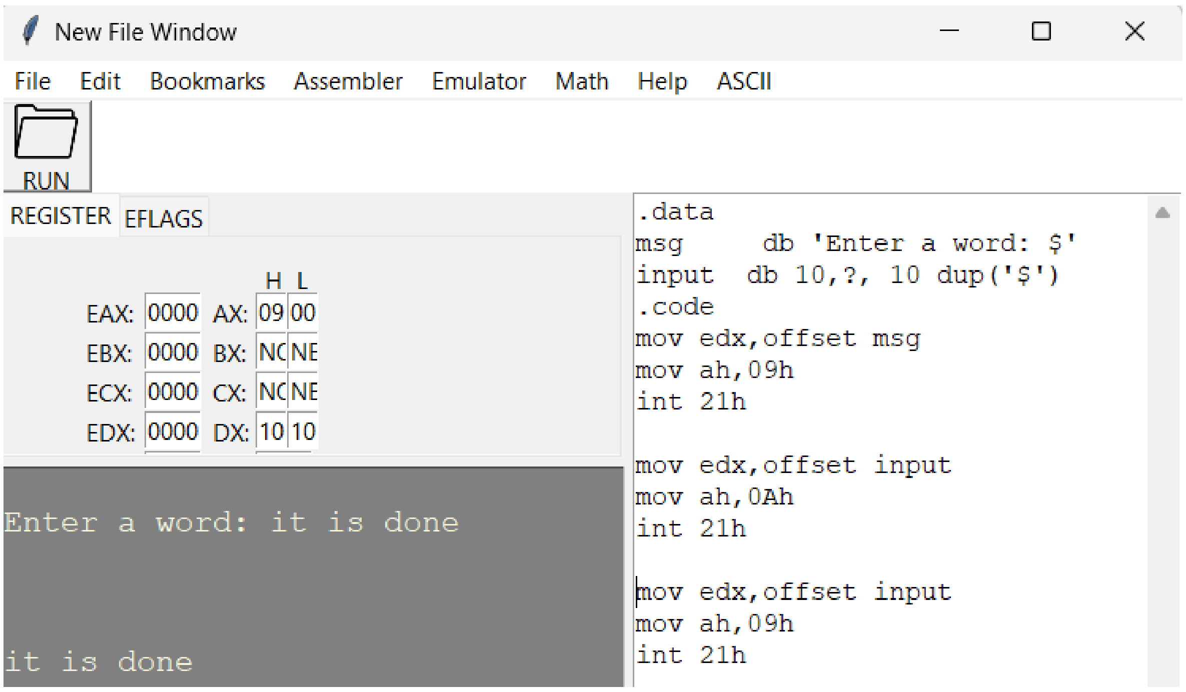
input db 10,?, 10 dup('$'): This line defines an array to store the word the user entered. Thefirst byte holds the number of characters the user entered. The second byte is a placeholderand may not be used. The next bytes make up the memory space reserved for the user's
input. 10 dup('$') reserves 10 $ characters in memory.
mov edx, offset input: This line loads the memory address of the input tag into the edx register. It points to the area of memory used to store the word that the user enters.

mov ah, 0Ah: This line loads the value 0Ah into the ah register. This value specifies the function number to be used in the call to int 21h. 0Ah calls this interrupt vector to read astring from the keyboard.

int 21h: This line reads the word entered by the user and stores it in the memory areapointed to by the input tag.

We did what we wanted, but if you notice, all register values became 'NONE' and this is asituation that should not happen.

For better testing, let's call our interrupt by assigning the 09H value we used to print this message to AH and test whether the string we entered is printed back to the console. When we do this we get the result in Figure 9.1.5. In other words, what we wanted did not happenand we did not get a different result from the previous one.



```
          H  L
EAX: 0000 AX: 09 00
EBX: 0000 BX: NC NE
ECX: 0000 CX: NC NE
EDX: 0000 DX: 10 10
```

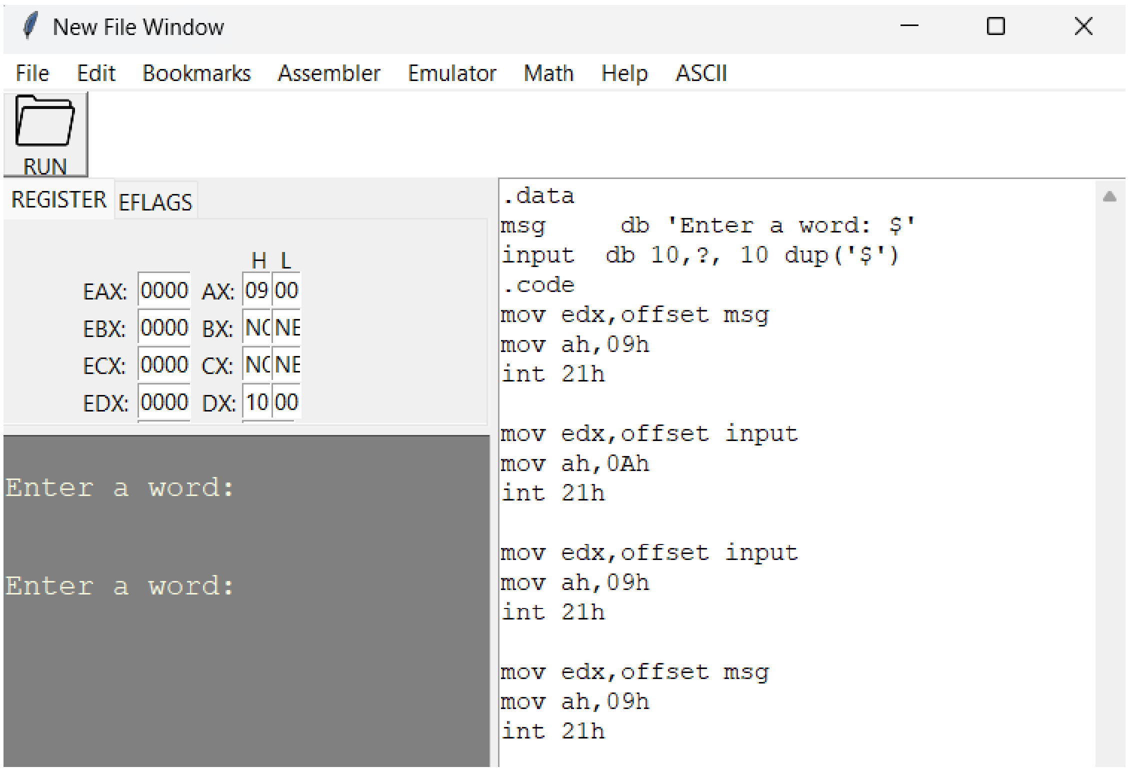
```
Enter a word: it is done

it is done
```

```
.data
msg      db 'Enter a word: $'
input  db 10,?, 10 dup('$')
.code
mov edx,offset msg
mov ah,09h
int 21h

mov edx,offset input
mov ah,0Ah
int 21h

mov edx,offset input
mov ah,09h
int 21h
```

Figure 9.7. Printing input

To better understand our error, let's try printing the initial msg (Enter a word:) again afterreceiving input from the user.

To achieve this, we can reprint the string in the msg variable by assigning the offset value ofinput to edx instead of the offset value of msg.



```
          H  L
EAX: 0000 AX: 09 00
EBX: 0000 BX: NC NE
ECX: 0000 CX: NC NE
EDX: 0000 DX: 10 00
```

```
Enter a word:

Enter a word:
```

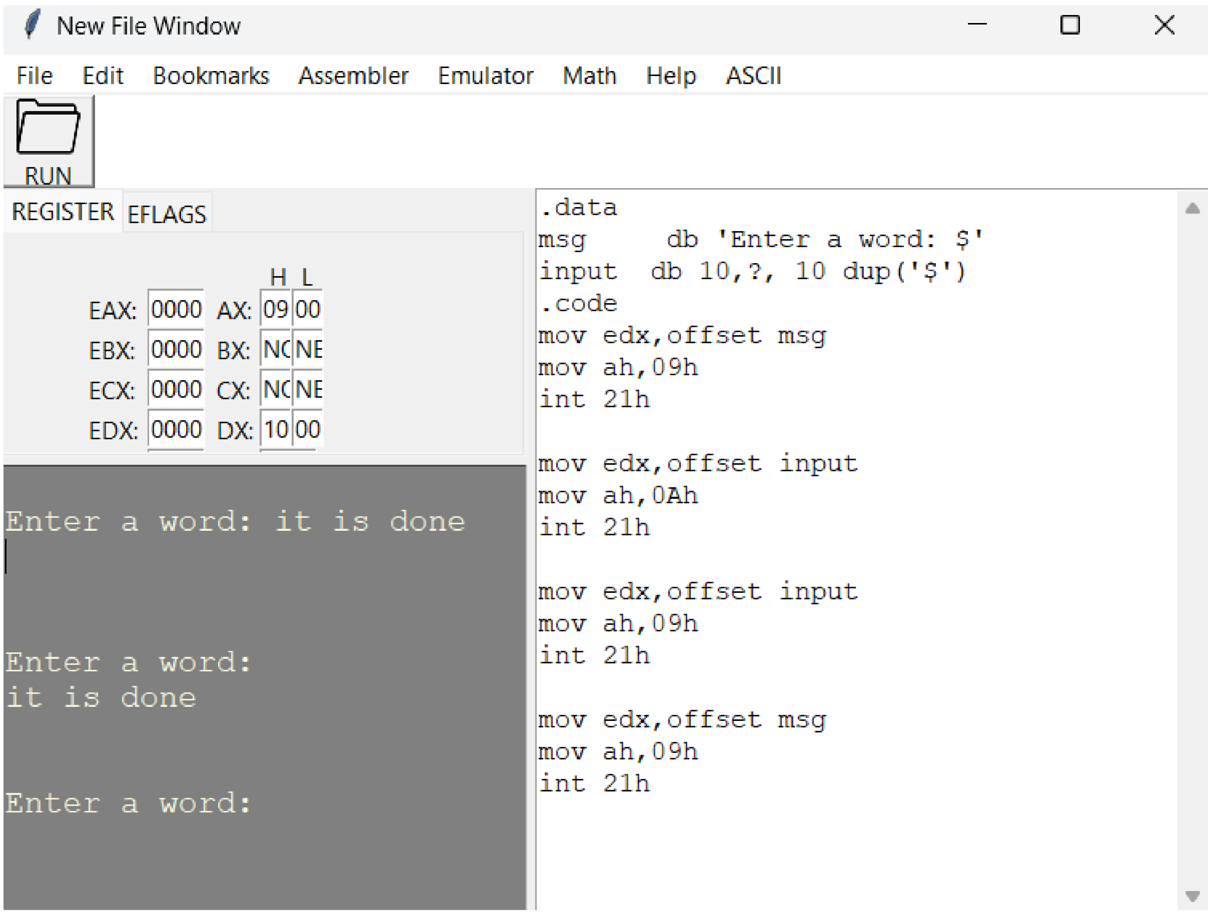
```
.data
msg      db 'Enter a word: $'
input  db 10,?, 10 dup('$')
.code
mov edx,offset msg
mov ah,09h
int 21h

mov edx,offset input
mov ah,0Ah
int 21h

mov edx,offset input
mov ah,09h
int 21h

mov edx,offset msg
mov ah,09h
int 21h
```

Figure 9.8. More detailed test 1

When we look at Figure 9.1.6, we can see that the phrase "Enter a word:" is printed twice,when it should have been printed once at first and waited for the user's input, so the codegoes to the next step without waiting for input.

To prevent this, we tried returning the index from the corresponding function, but this only restarted the script at that index after the input was entered. Looking at Figure 9.1.7, we seethat it prints msg twice again. Then when we enter the input, this input is printed and the msg statement "Enter a word:" is printed again.

So the problem is that the script does all the operations in the script before waiting for input.The index rotation solution only made the code work correctly after that operation.



```
     H  L
EAX: 0000 AX: 09 00
EBX: 0000 BX: NC NE
ECX: 0000 CX: NC NE
EDX: 0000 DX: 10 00
```

```
Enter a word: it is done


Enter a word:
it is done


Enter a word:
```

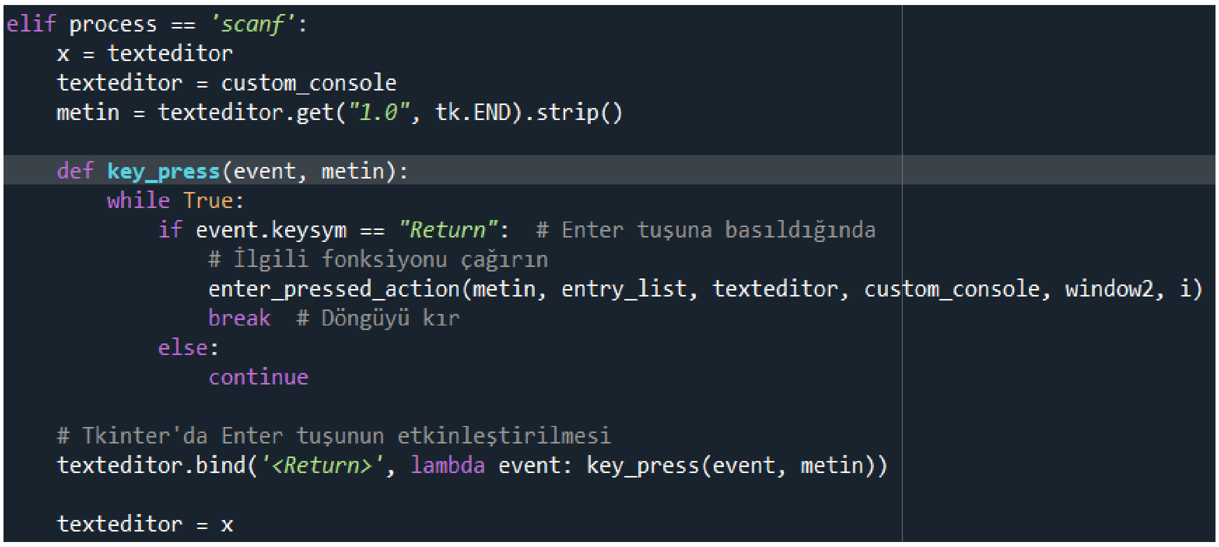
```
.data
msg      db 'Enter a word: $'
input  db 10,?, 10 dup('$')
.code
mov edx,offset msg
mov ah,09h
int 21h

mov edx,offset input
mov ah,0Ah
int 21h

mov edx,offset input
mov ah,09h
int 21h

mov edx,offset msg
mov ah,09h
int 21h
```

Figure 9.9. More detailed test 2

You can see the relevant Python code in figure 9.1.8. Here, the code comes here if you want to do a scanf operation, that is, if you want to get input.

In this code, the function 'def key_press(event, text)' checks if the 'enter' key is pressed. Here, with the help of an infinite loop, it is aimed to repeat the loop if the 'enter' key is notpressed. When the 'enter' key is pressed, the loop is broken with the 'break' command.

We thought this should solve our problem, but even though it is an infinite loop, the codeperforms other operations even if the 'enter' key is not pressed. This could not be avoidedand EMUPENT failed in this part.



```
elif process == 'scanf':
    x = texteditor
    texteditor = custom_console
    metin = texteditor.get("1.0", tk.END).strip()

    def key_press(event, metin):
        while True:
            if event.keysym == "Return":  # Enter tuşuna basıldığında
                # İlgili fonksiyonu çağırın
                enter_pressed_action(metin, entry_list, texteditor, custom_console, window2, i)
                break  # Döngüyü kır
            else:
                continue

    # Tkinter'da Enter tuşunun etkinleştirilmesi
    texteditor.bind('<Return>', lambda event: key_press(event, metin))

    texteditor = x
```

Figure 9.10. Code for getting input

9.1.2. EVALUATION

If we talk about the strengths of EMUPENT's console; we can get the output we want in operations with registers, we can print our error messages, we can print the string we want.

If we talk about the weaknesses of EMUPENT's console; we get string input, but the code does not work as desired and gives different errors. Since this is experienced in string input, problems may occur in the future when other inputs such as mouse input are desired to bereceived. For these reasons, we need to find a new solution and design a perfect console bycorrecting the weak parts of the console.

9.2. PYTHON CONSOLE

We had integrated the console into our interface using the tkinter library. However, the stdout function in the sys library does not support working with the input() function, and theget() function does not work as desired. We worked for 3-4 weeks on this issue, even trying to manually write functions from the console (such as receiving data when the enter key is pressed). However, we realized that this could potentially cause problems. As atemporary solution, we decided to work on Python's own console. Technically, this is not the correct approach.

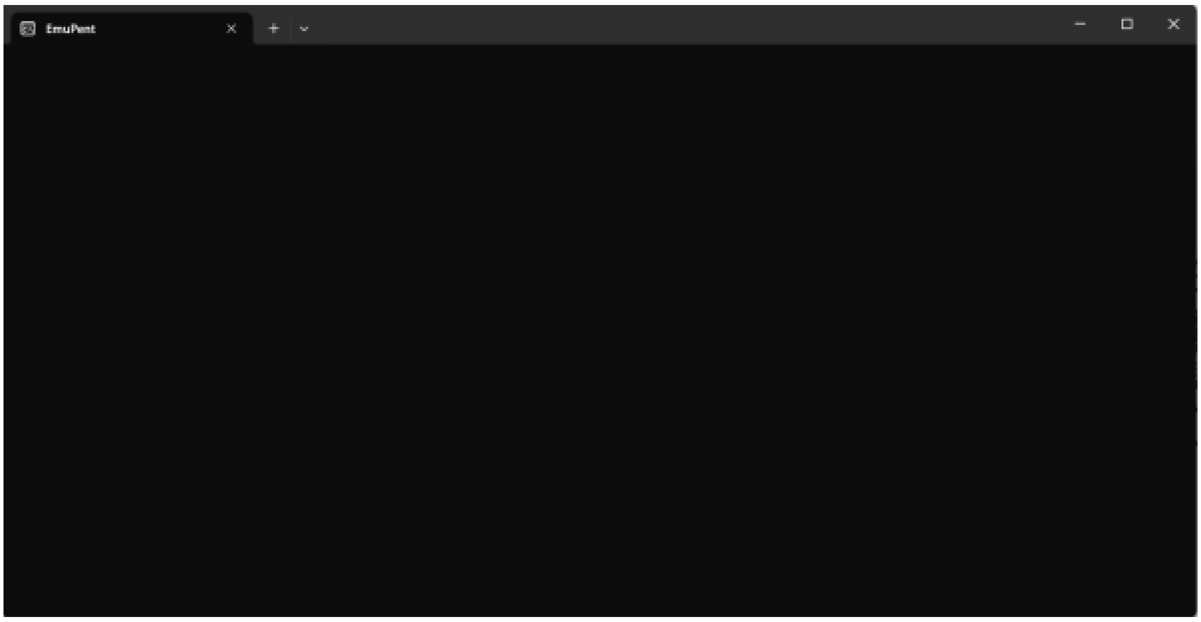
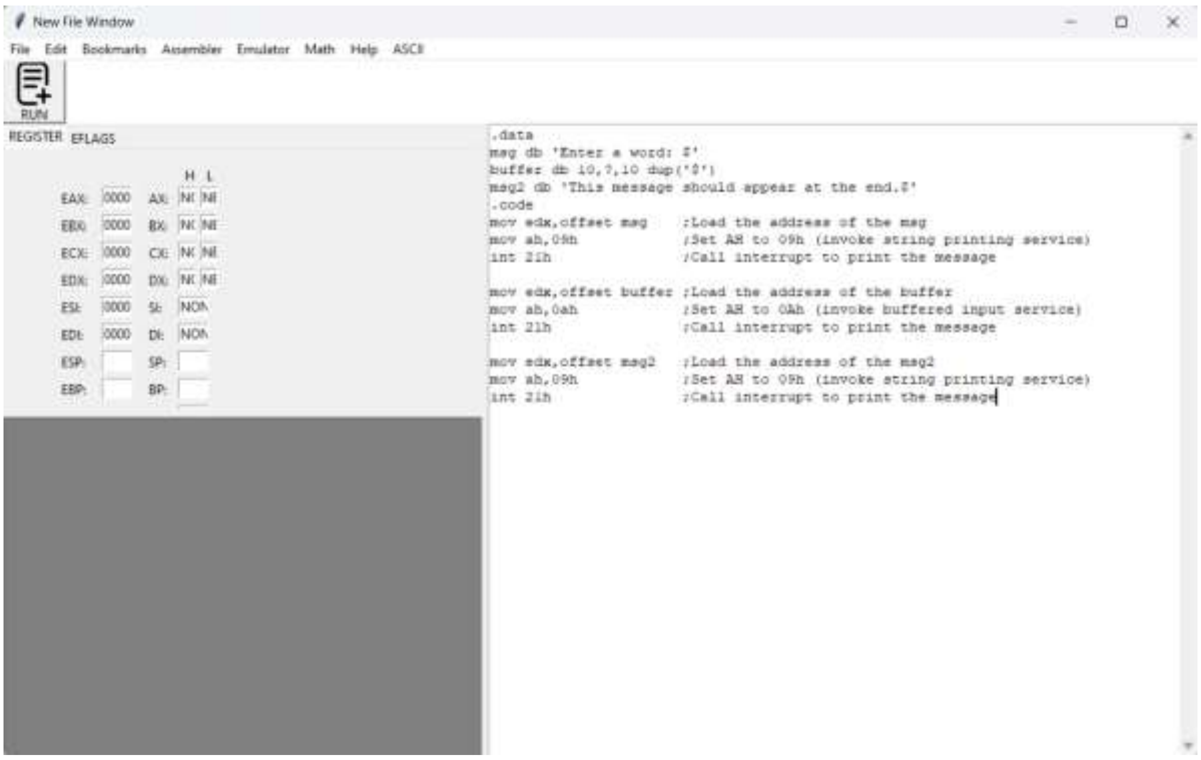


Figure 9.11. New console of EMUPENT

9.2.1. TESTING AND RESULTS

In the console we explained in the previous section, all operations were performed without waiting for input. In other words, the message that should be printed at the end was printed at the beginning or the registers that should be set at the end were set at the beginning.

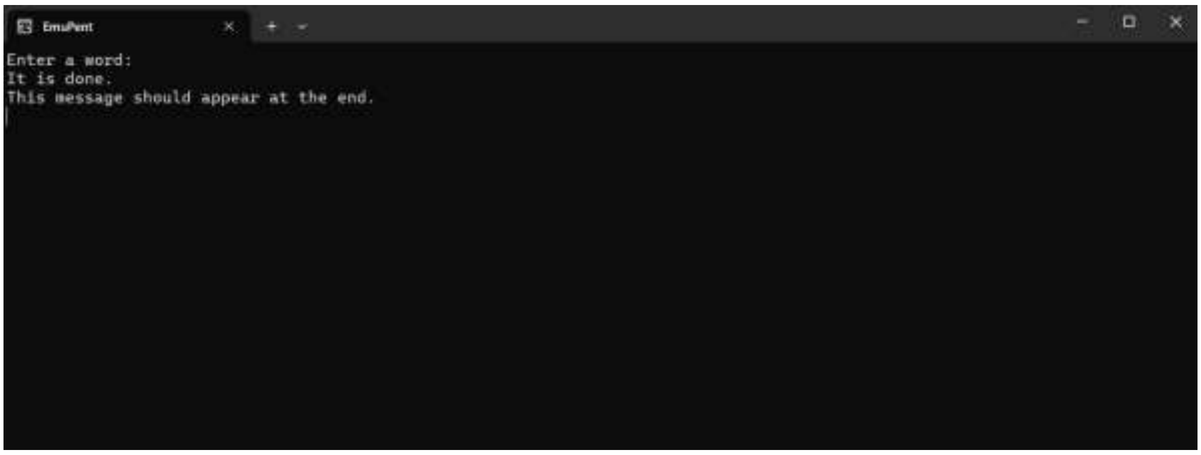
In the assembly code that you can see in figure 9.2.1 for our new console, first 'Enter a word:' will be printed and input from the user will be expected. When the input is entered and enter is pressed, 'This message should appear at the end.' will be printed.



```
.data
msg db 'Enter a word: $'
buffer db 10,?,10 dup('$')
msg2 db 'This message should appear at the end.$'
.code
mov edx,offset msg    ;Load the address of the msg
mov ah,09h            ;Set AH to 09h (invoke string printing service)
int 21h               ;Call interrupt to print the message

mov edx,offset buffer ;Load the address of the buffer
mov ah,0ah            ;Set AH to 0Ah (invoke buffered input service)
int 21h               ;Call interrupt to print the message

mov edx,offset msg2   ;Load the address of the msg2
mov ah,09h            ;Set AH to 09h (invoke string printing service)
int 21h               ;Call interrupt to print the message
```

Figure 9.12. Assembly code for new console

When we run the code and type 'It is done' and press enter:



```
Enter a word:
It is done.
This message should appear at the end.
```

Figure 9.13. Console Screen for trial

9.2.2. EVALUATION

EMUPENT's current console is a corrected version of the weaknesses of that console on top of what it could do in the previous part, namely console in interface.

But this console also has a weakness. The weakness is that when we first run EMUPENT, the console opens directly. In other words, when we enter an assembly code in our text editor and press run, the console should open, but the console is always open. Also, if we close the console, the programme closes completely.

10. FINAL DESIGN

10.1. FINAL REVIEW AND EXPERIMENT

In the seventh chapter, I had created a flowchart about how the buttons in the first interface work in Figure 7.5. We couldn't perform all the operations in the flowchart created at the end of the project. The Open Last and Create a Flowchart buttons are not working actively. Here is our Final Flowchart:

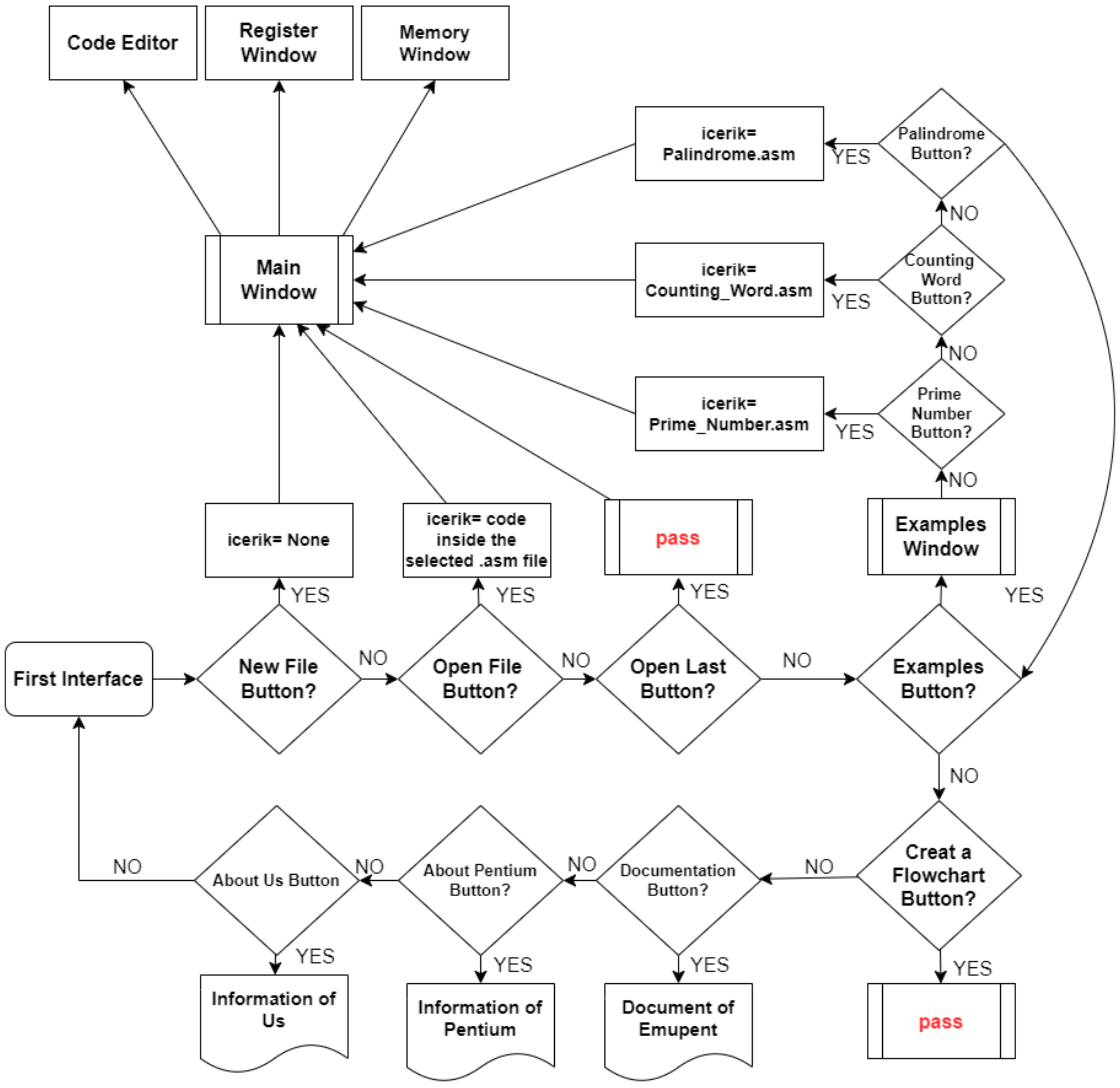


Figure 10.1. Flowchart of final project (The places shown in red are places that differ from Figure 7.1)

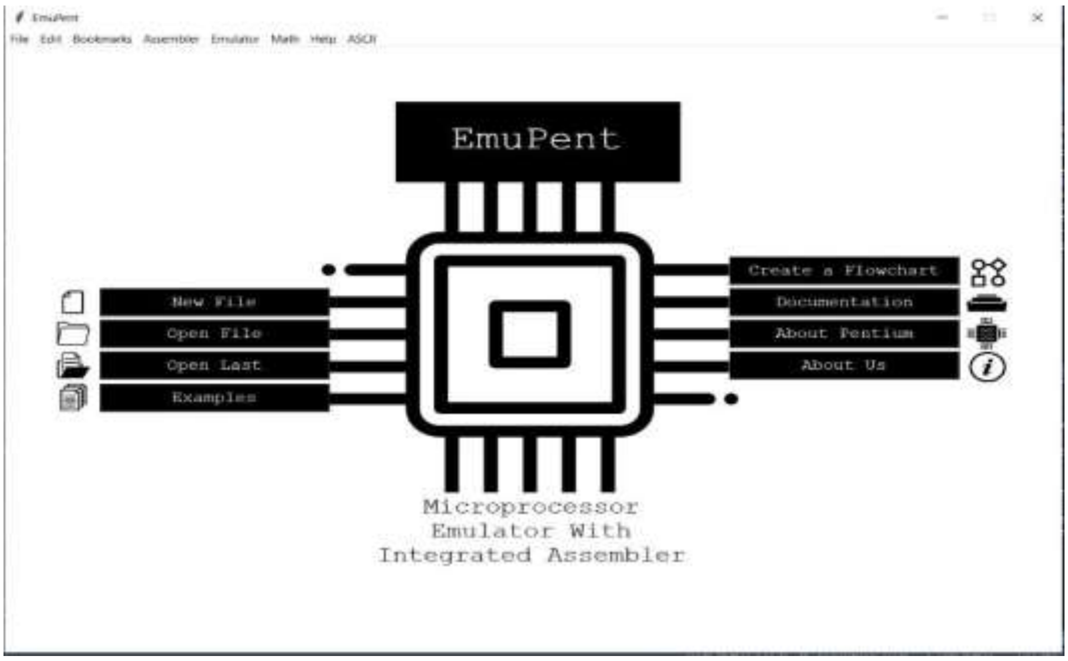
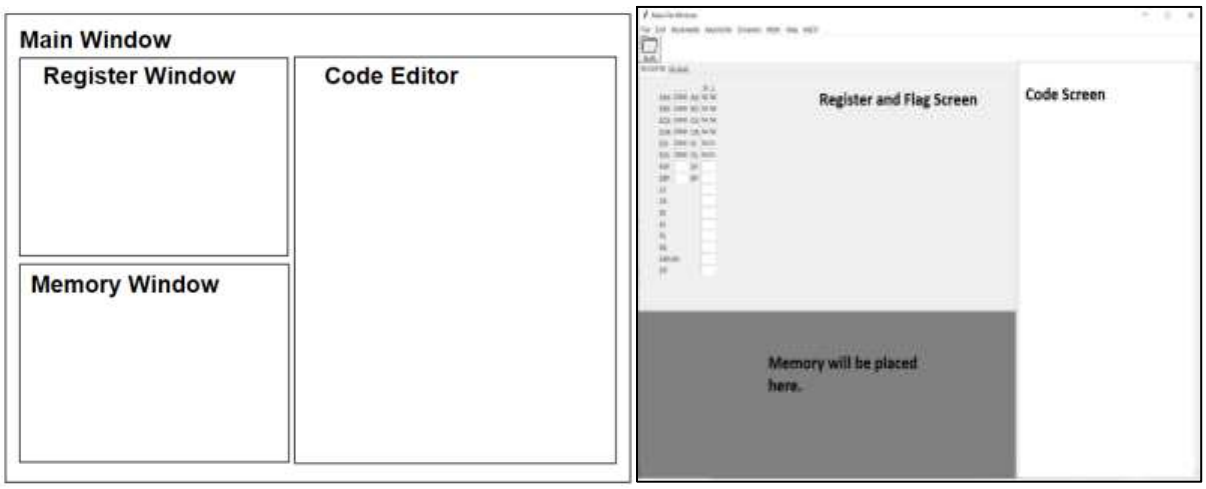


Figure 10.2. First Interface

When the New File Button is pressed, the main window will open. The general topology of the main window planned in Chapter 7 is compatible with the final version of our emulator that we created with Figure 7.4.



(a) (b)

Figure 10.3. (a) Planned Main window (b)Final Main Window

Now, let's explain the parts of this main window one by one. The operation of the Main Window works exactly as planned with the flowchart in 7.5.

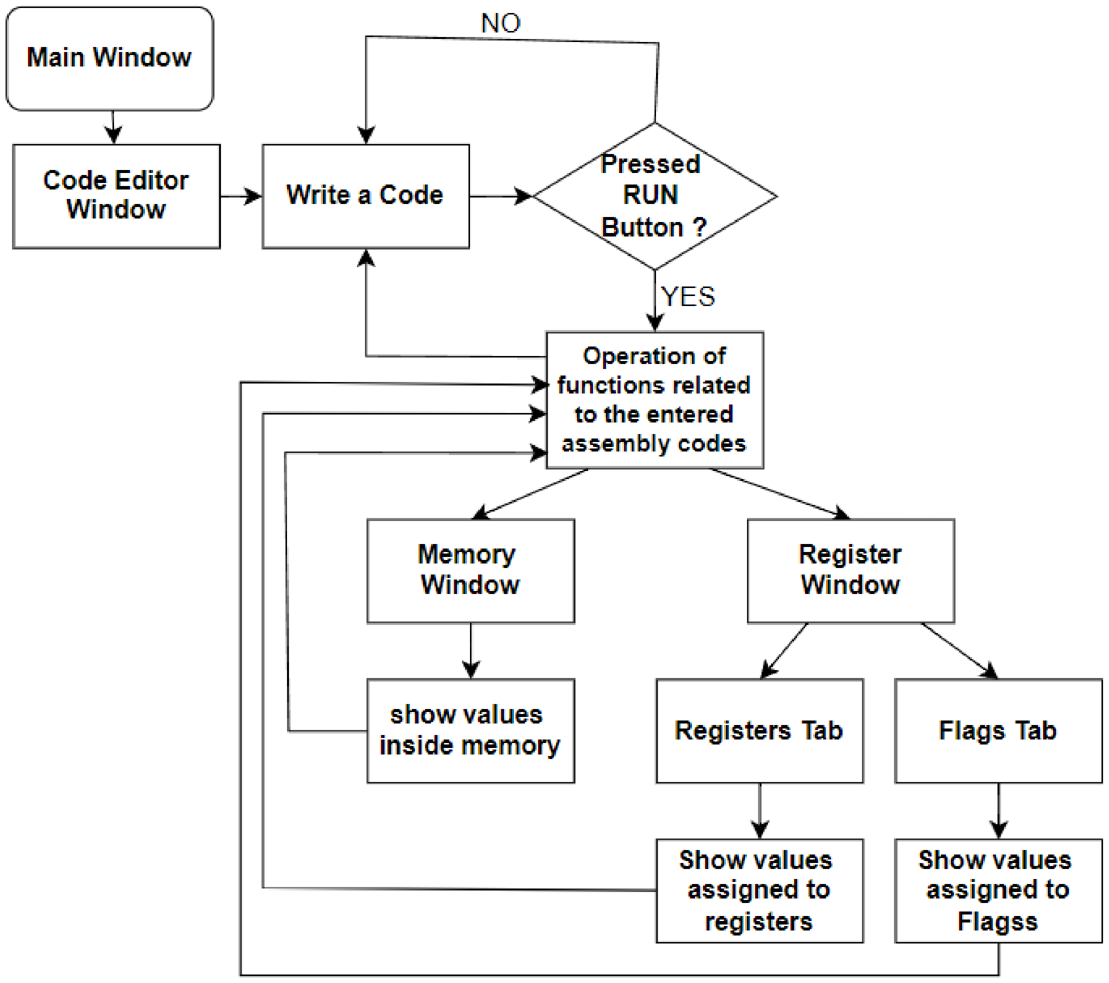
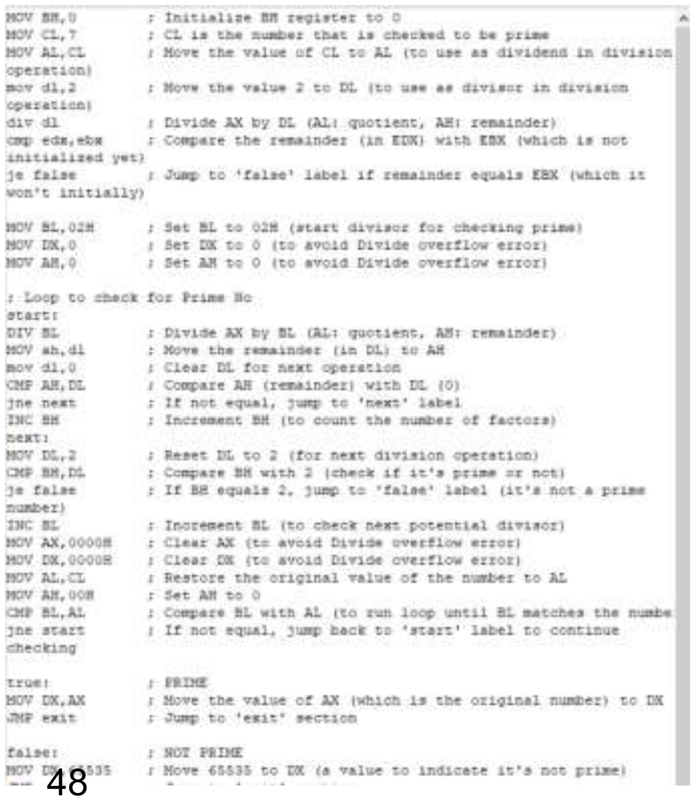

Main Window
Code Editor Window
Write a Code
NO
Pressed RUN Button ?
YES
Operation of functions related to the entered assembly codes
Memory Window
Register Window
show values inside memory
Registers Tab
Flags Tab
Show values assigned to registers
Show values assigned to Flagss

Figure 10.4. The flowchart of the main window’s operation.

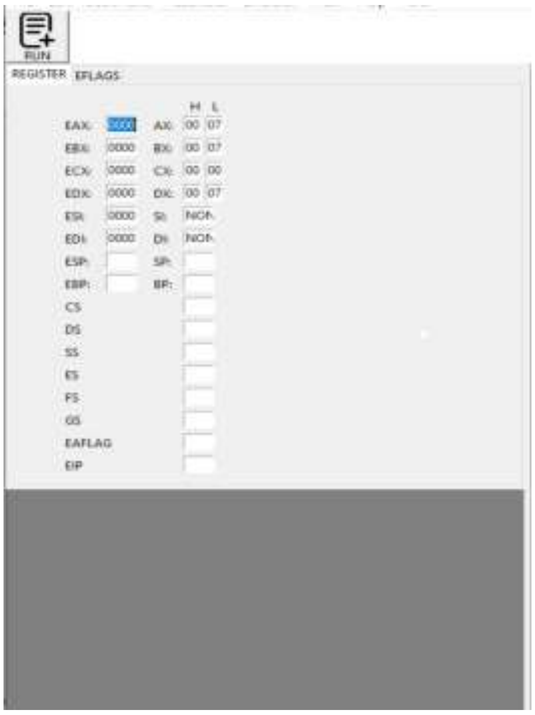
10.1.1. CODE EDITOR

You can write all the commands from our final report here and process these commands in EmuPent by pressing the RUN button or the F5 key. There is no case distinction in our code editor. You can write commands in either uppercase or lowercase as you wish. Thanks to the scroll created on the right, you can enter code of any length.



```
MOV BH,0          ; Initialize BH register to 0
MOV CL,7          ; CL is the number that is checked to be prime
MOV AL,CL         ; Move the value of CL to AL (to use as dividend in division
operation)
mov dl,2          ; Move the value 2 to DL (to use as divisor in division
operation)
div dl            ; Divide AX by DL (AL: quotient, AH: remainder)
cmp edx,ebx       ; Compare the remainder (in EDX) with EBX (which is not
initialized yet)
je false          ; Jump to 'false' label if remainder equals EBX (which it
won't initially)

MOV BL,02H        ; Set BL to 02H (start divisor for checking prime)
MOV DX,0          ; Set DX to 0 (to avoid Divide overflow error)
MOV AH,0          ; Set AH to 0 (to avoid Divide overflow error)

; Loop to check for Prime No
start:
DIV BL            ; Divide AX by BL (AL: quotient, AH: remainder)
MOV ah,dl         ; Move the remainder (in DL) to AH
mov dl,0          ; Clear DL for next operation
CMP AH,DL         ; Compare AH (remainder) with DL (0)
jne next          ; If not equal, jump to 'next' label
INC BH            ; Increment BH (to count the number of factors)
next:
MOV DL,2          ; Reset DL to 2 (for next division operation)
CMP BH,DL         ; Compare BH with 2 (check if it's prime or not)
je false          ; If BH equals 2, jump to 'false' label (it's not a prime
number)
INC BL            ; Increment BL (to check next potential divisor)
MOV AX,0000H      ; Clear AX (to avoid Divide overflow error)
MOV DX,0000H      ; Clear DX (to avoid Divide overflow error)
MOV AL,CL         ; Restore the original value of the number to AL
MOV AH,00H        ; Set AH to 0
CMP BL,AL         ; Compare BL with AL (to run loop until BL matches the numbe
jne start         ; If not equal, jump back to 'start' label to continue
checking

true:             ; PRIME
MOV DX,AX         ; Move the value of AX (which is the original number) to DX
JMP exit          ; Jump to 'exit' section

false:            ; NOT PRIME
MOV DX,65535      ; Move 65535 to DX (a value to indicate it's not prime)
```

Figure 10.4. Code Editor



```
                    H  L
EAX: 0000   AX:    00 07
EBX: 0000   BX:    00 07
ECX: 0000   CX:    00 00
EDX: 0000   DX:    00 07
ESI: 0000   SI:    NON
EDI: 0000   DI:    NON
ESP:        SP:
EBP:        BP:
CS
DS
SS
ES
FS
GS
EAFLAG
EIP
```

10.1.2. REGISTERS & FLAGS WINDOW

In this section, we can see the output of the commands processed in the emulator in the registers.

10.1.3. MEMORY WINDOW

When a string operation is performed, we can see the ASCII code of the characters assigned to the address values we assigned with esi, edi from here.

Here, we explained the basic operations in the main window, now let's move on to more specific prefixes. Within the scope of this project, we tried to conduct 4 experiments. Now, let's do these experiments one by one and examine the results we obtained.

Figure 10.5. Windows on main window

10.2. EXPEIMENTS

10.2.1. PRIME NUMBER EXPERIMENT

Numbers that are only divisible by themselves and 1 are called prime numbers. In this experiment, we use assembly code to check whether the value assigned to the CL register is a prime number or not.

10.2.1.1. CODE

A prime number (or a prime) is a natural number greater than 1 that is not a product of two smaller natural numbers. A natural number greater than 1 that is not prime is called a composite number.

```
MOV BH,0        ; Initialize BH register to 0
MOV CL,7        ; CL is the number that is checked to be prime
MOV AL,CL       ; Move the value of CL to AL (to use as dividend in
division operation)
                    ; Checking for a prime number 2
mov dl,2        ; Move the value 2 to DL (to use as divisor in
division operation)
div dl          ; Divide AX by DL (AL: quotient, AH: remainder)
cmp edx,ebx     ; Compare the remainder (in EDX) with EBX (which is
not initialized yet)
je false        ; Jump to 'false' label if remainder equals EBX
(which it won't initially)
MOV BL,02H      ; Set BL to 02H (start divisor for checking prime)
MOV DX,0        ; Set DX to 0 (to avoid Divide overflow error)
MOV AH,0          ; Set AH to 0 (to avoid Divide overflow error)
 ;Loop to check for Prime No
 start:
 DIV BL            ; Divide AX by BL (AL: quotient, AH: remainder)
 MOV ah,dl         ; Move the remainder (in DL) to AH
 mov dl,0          ; Clear DL for next operation
 CMP AH,DL         ; Compare AH (remainder) with DL (0)
 jne next          ; If not equal, jump to 'next' label
 INC BH            ; Increment BH (to count the number of factors)
 next:
 MOV DL,2          ; Reset DL to 2 (for next division operation)
 CMP BH,DL         ; Compare BH with 2 (check if it's prime or not)
 je false          ; If BH equals 2, jump to 'false' label (it's not a
 prime number)
 INC BL            ; Increment BL (to check next potential divisor)
 MOV AX,0000H      ; Clear AX (to avoid Divide overflow error)
 MOV DX,0000H      ; Clear DX (to avoid Divide overflow error)
 MOV AL,CL         ; Restore the original value of the number to AL
 MOV AH,00H        ; Set AH to 0
 CMP BL,AL         ; Compare BL with AL (to run loop until BL matches
 the number)
 jne start         ; If not equal, jump back to 'start' label to
 continue checking

 true:             ; PRIME
 MOV DX,AX         ; Move the value of AX (which is the original
 number) to DX
 JMP exit          ; Jump to 'exit' section
 false:               ; NOT PRIME
 MOV DX,65535      ; Move 65535 to DX (a value to indicate it's not
 prime)
 JMP exit          ; Jump to 'exit' section
 exit:
 MOV AL,CL         ; Move the original value of the number back to AL
 MOV CL,0          ; Clear CL
```

10.2.1.2. FLOWCHART OF CODE

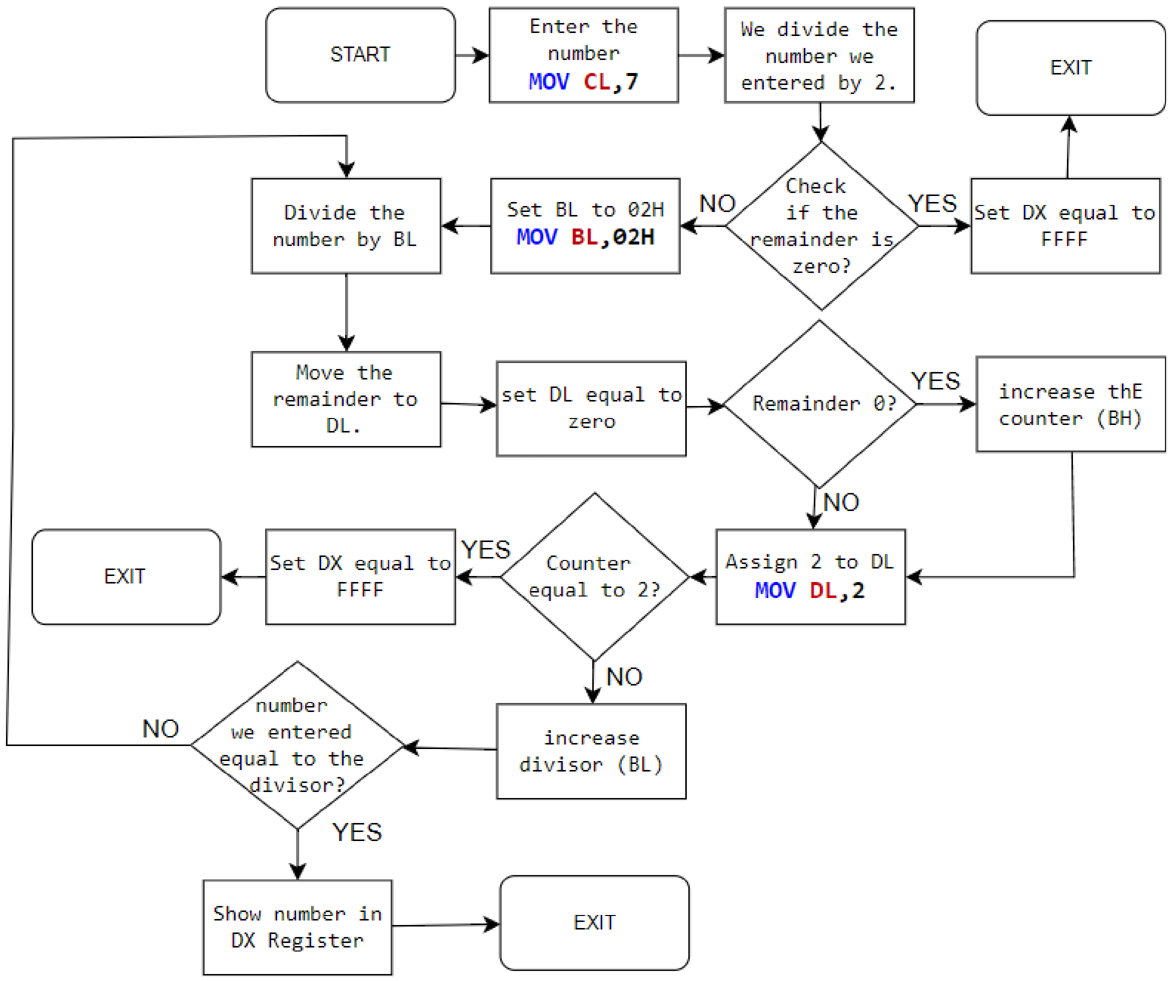
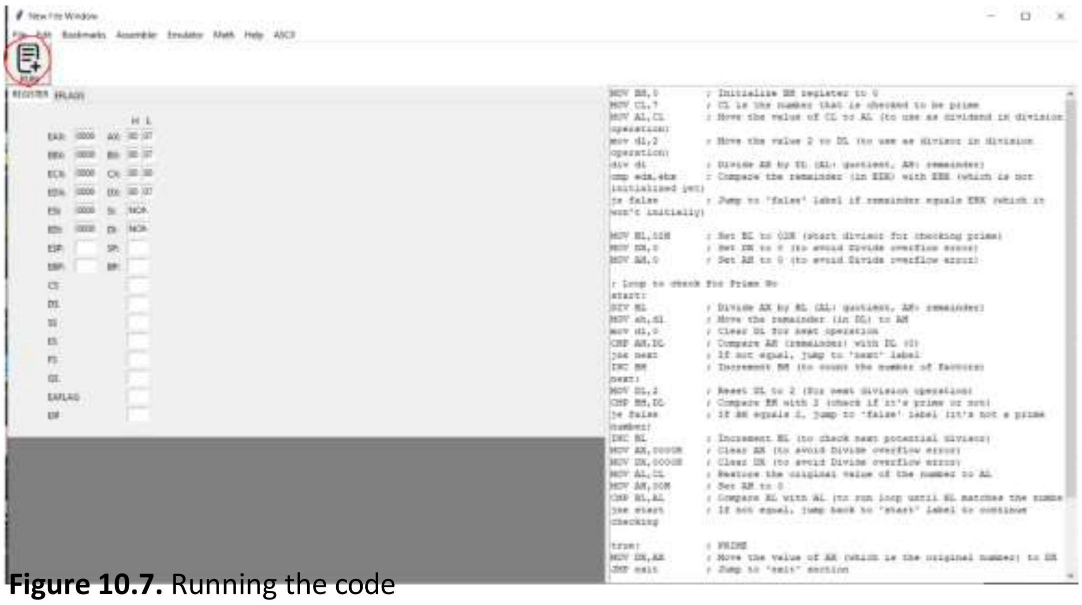


Figure 10.6. Flowchart of Prime Number Experiment

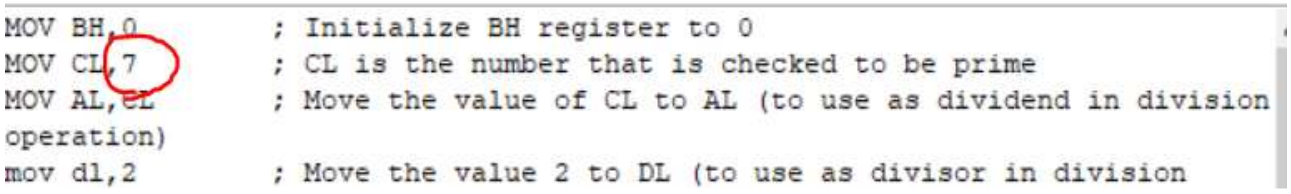
10.2.1.3. PROCESSING OF CODE



```
MOV BH,0          ; Initialize BH register to 0
MOV CL,7          ; CL is the number that is checked to be prime
MOV AL,CL         ; Move the value of CL to AL (to use as dividend in division
operation)
mov dl,2          ; Move the value 2 to DL (to use as divisor in division
operation)
div dl            ; Divide AX by DL (AL: quotient, AH: remainder)
cmp edx,ebx       ; Compare the remainder (in EDX) with EBX (which is not
initialized yet)
je false          ; Jump to 'false' label if remainder equals EBX (which it
won't initially)

MOV BL,02H        ; Set BL to 02H (start divisor for checking prime)
MOV DX,0          ; Set DX to 0 (to avoid Divide overflow error)
MOV AH,0          ; Set AH to 0 (to avoid Divide overflow error)

; Loop to check for Prime No
start:
DIV BL            ; Divide AX by BL (AL: quotient, AH: remainder)
MOV ah,dl         ; Move the remainder (in DL) to AH
mov dl,0          ; Clear DL for next operation
CMP AH,DL         ; Compare AH (remainder) with DL (0)
jne next          ; If not equal, jump to 'next' label
INC BH            ; Increment BH (to count the number of factors)
next:
MOV DL,2          ; Reset DL to 2 (for next division operation)
CMP BH,DL         ; Compare BH with 2 (check if it's prime or not)
je false          ; If BH equals 2, jump to 'false' label (it's not a prime
number)
INC BL            ; Increment BL (to check next potential divisor)
MOV AX,0000H      ; Clear AX (to avoid Divide overflow error)
MOV DX,0000H      ; Clear DX (to avoid Divide overflow error)
MOV AL,CL         ; Restore the original value of the number to AL
MOV AH,00H        ; Set AH to 0
CMP BL,AL         ; Compare BL with AL (to run loop until BL matches the numbe
jne start         ; If not equal, jump back to 'start' label to continue
checking

true:             ; PRIME
MOV DX,AX         ; Move the value of AX (which is the original number) to DX
JMP exit          ; Jump to 'exit' section
```

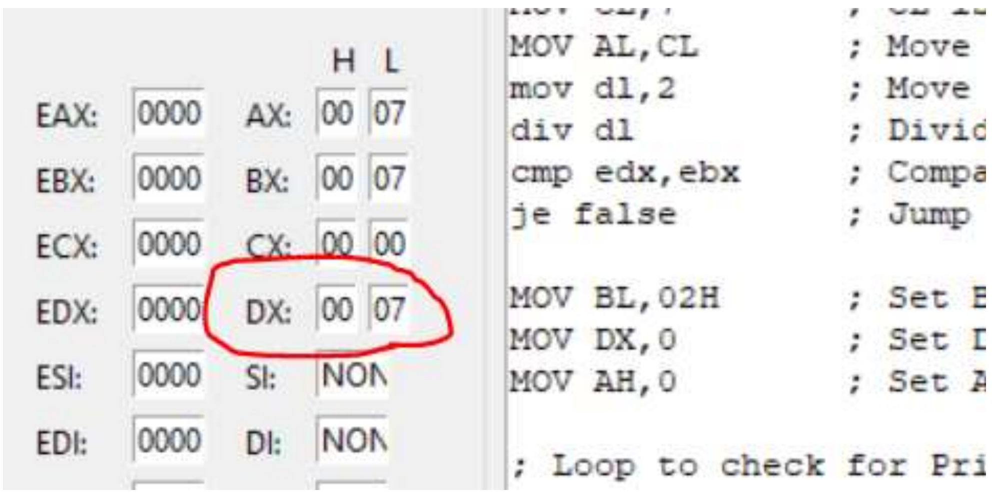
Figure 10.7. Running the code

We can check the number by changing the leading CL value in the code.
When Enter the 7.



```
MOV BH,0          ; Initialize BH register to 0
MOV CL,7          ; CL is the number that is checked to be prime
MOV AL,CL         ; Move the value of CL to AL (to use as dividend in division
operation)
mov dl,2          ; Move the value 2 to DL (to use as divisor in division
```

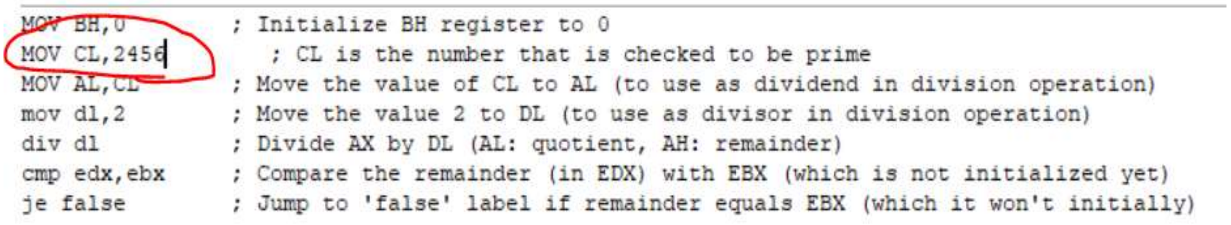
Figure 10.8. Prime number example 1.



```
EAX: 0000   AX: 00 07        MOV AL,CL       ; Move
EBX: 0000   BX: 00 07        mov dl,2        ; Move
ECX: 0000   CX: 00 00        div dl          ; Divic
EDX: 0000   DX: 00 07        cmp edx,ebx     ; Compa
ESI: 0000   SI: NON          je false        ; Jump
EDI: 0000   DI: NON
                             MOV BL,02H      ; Set E
                             MOV DX,0        ; Set I
                             MOV AH,0        ; Set A

                             ; Loop to check for Pri
```

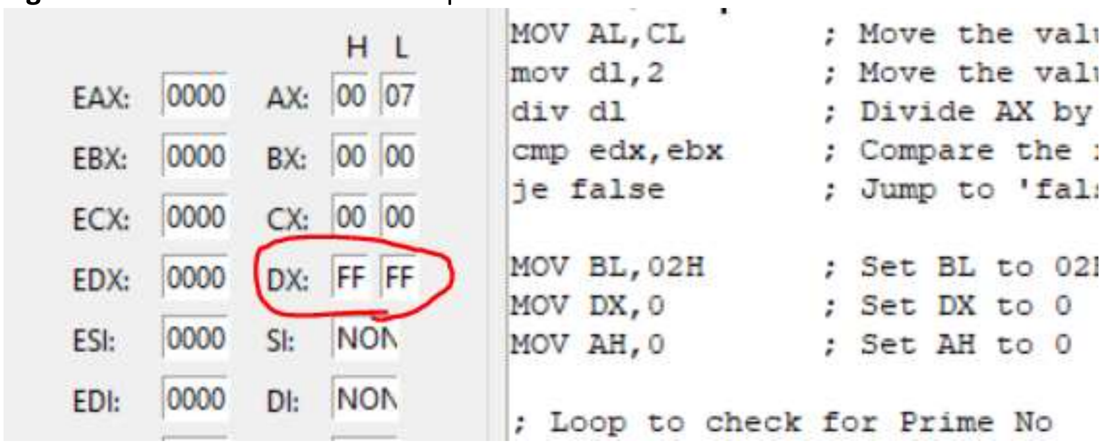
Figure 10.9. Result of example 1.

When enter the 2456.



```
MOV BH,0          ; Initialize BH register to 0
MOV CL,2456         ; CL is the number that is checked to be prime
MOV AL,CL         ; Move the value of CL to AL (to use as dividend in division operation)
mov dl,2          ; Move the value 2 to DL (to use as divisor in division operation)
div dl            ; Divide AX by DL (AL: quotient, AH: remainder)
cmp edx,ebx       ; Compare the remainder (in EDX) with EBX (which is not initialized yet)
je false          ; Jump to 'false' label if remainder equals EBX (which it won't initially)
```

Figure 10.10. Prime number example 2.



```
EAX: 0000   AX: 00 07        MOV AL,CL       ; Move the valu
EBX: 0000   BX: 00 00        mov dl,2        ; Move the valu
ECX: 0000   CX: 00 00        div dl          ; Divide AX by
EDX: 0000   DX: FF FF        cmp edx,ebx     ; Compare the :
ESI: 0000   SI: NON          je false        ; Jump to 'fal:
EDI: 0000   DI: NON
                             MOV BL,02H      ; Set BL to 02I
                             MOV DX,0        ; Set DX to 0
                             MOV AH,0        ; Set AH to 0

                             ; Loop to check for Prime No
```

Figure 10.11. Result of experiment

10.2.2. COUNTING EXPERIMENT

In this experiment we will find how many letters a string defined in the .data section has. In the .data section, a string is defined in the msg variable and in the .code section, we load the memory address of the variable msg into the edx register with the help of offset. In this way, string related operations can be done.

10.2.2.1. CODE

```
.data
msg db 'Technically, anything with a microprocessor can be
considered a robot.$'
.code
mov edx,offset msg ; Load the address of the message into EDX
mov ah,09h ; Set AH to 09h (invoke string printing service)
int 21h ; Call interrupt to print the message
mov ebx,0 ; Initialize EBX to 0 (used as a comparison value)
mov ecx,1 ; Initialize ECX to 1 (used as a counter)
loop:
mov eax,[esi] ; Load the value at ESI into EAX
inc esi ; Increment ESI to point to the next character
cmp eax,20h ; Compare EAX with the ASCII value of a space (' ')
je pluss ; If equal, jump to pluss (increment the counter)
cmp eax,24h ; Compare EAX with the ASCII value of '$' (end of
message)
cmp eax,ebx ; Compare EAX with the value in EBX (0)
je decision ; If equal, jump to decision
jmp loop ; Otherwise, repeat the loop
pluss:
inc ecx; Increment the counter
jmp loop ; Continue the loop
decision:
mov ebx,9 ; Set EBX to 9 for comparison
cmp ebx,ecx ; Compare EBX with ECX (counter)
jne over_9 ; If not equal, jump to over_9
mov eax,30h ; Load the ASCII value of '0' into EAX
add ecx,eax ; Convert the counter to its ASCII value
mov edx,ecx ; Move the ASCII value to EDX for printing
mov ah,02h ; Set AH to 02h (invoke character printing service)
int 21h ; Call interrupt to print the character
over_9:
mov eax,ecx ; Move the counter value to EAX
mov ebx,10 ; Set EBX to 10 for division
div ebx ; Divide EAX by 10 (result in EAX, remainder in EDX)
mov edi,edx ; Move the remainder to EDI
mov edx,eax ; Move the quotient to EDX
mov eax,30h ; Load the ASCII value of '0' into EAX
add edx,eax ; Convert the quotient to its ASCII value
mov ah,02h ; Set AH to 02h (invoke character printing service)
```

```
int 21h ; Call interrupt to print the character
mov edx,edi ; Move the remainder (stored in EDI) to EDX
mov eax,30h ; Load the ASCII value of '0' into EAX
add edx,eax ; Convert the remainder to its ASCII value
mov ah,02h ; Set AH to 02h (invoke character printing service)
int 21h ; Call interrupt to print the character
```

10.2.2.2. FLOWCHART OF CODE

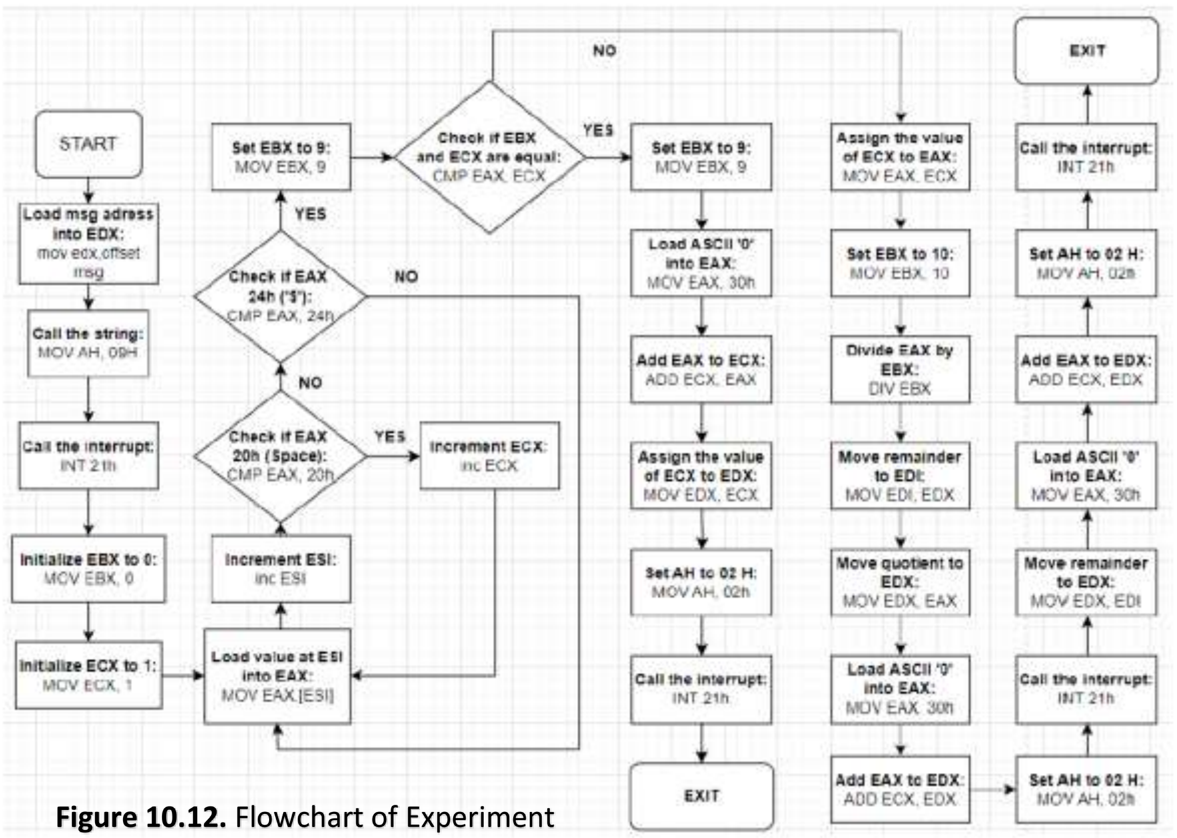

START
Load msg adress into EDX: mov edx,offset msg
Call the string: MOV AH, 09H
Call the interrupt: INT 21h
Initialize EBX to 0: MOV EBX, 0
Initialize ECX to 1: MOV ECX, 1
Load value at ESI into EAX: MOV EAX,[ESI]
Increment ESI: inc ESI
Check if EAX 20h (Space): CMP EAX, 20h
YES
Increment ECX: inc ECX
NO
Check if EAX 24h ('$'): CMP EAX, 24h
NO
YES
Set EBX to 9: MOV EBX, 9
Check if EBX and ECX are equal: CMP EAX, ECX
NO
YES
Set EBX to 9: MOV EBX, 9
Load ASCII '0' into EAX: MOV EAX, 30h
Add EAX to ECX: ADD ECX, EAX
Assign the value of ECX to EDX: MOV EDX, ECX
Set AH to 02 H: MOV AH, 02h
Call the interrupt: INT 21h
EXIT
Assign the value of ECX to EAX: MOV EAX, ECX
Set EBX to 10: MOV EBX, 10
Divide EAX by EBX: DIV EBX
Move remainder to EDI: MOV EDI, EDX
Move quotient to EDX: MOV EDX, EAX
Load ASCII '0' into EAX: MOV EAX, 30h
Add EAX to EDX: ADD ECX, EDX
Set AH to 02 H: MOV AH, 02h
Call the interrupt: INT 21h
Move remainder to EDX: MOV EDX, EDI
Load ASCII '0' into EAX: MOV EAX, 30h
Add EAX to EDX: ADD ECX, EDX
Set AH to 02 H: MOV AH, 02h
Call the interrupt: INT 21h
EXIT

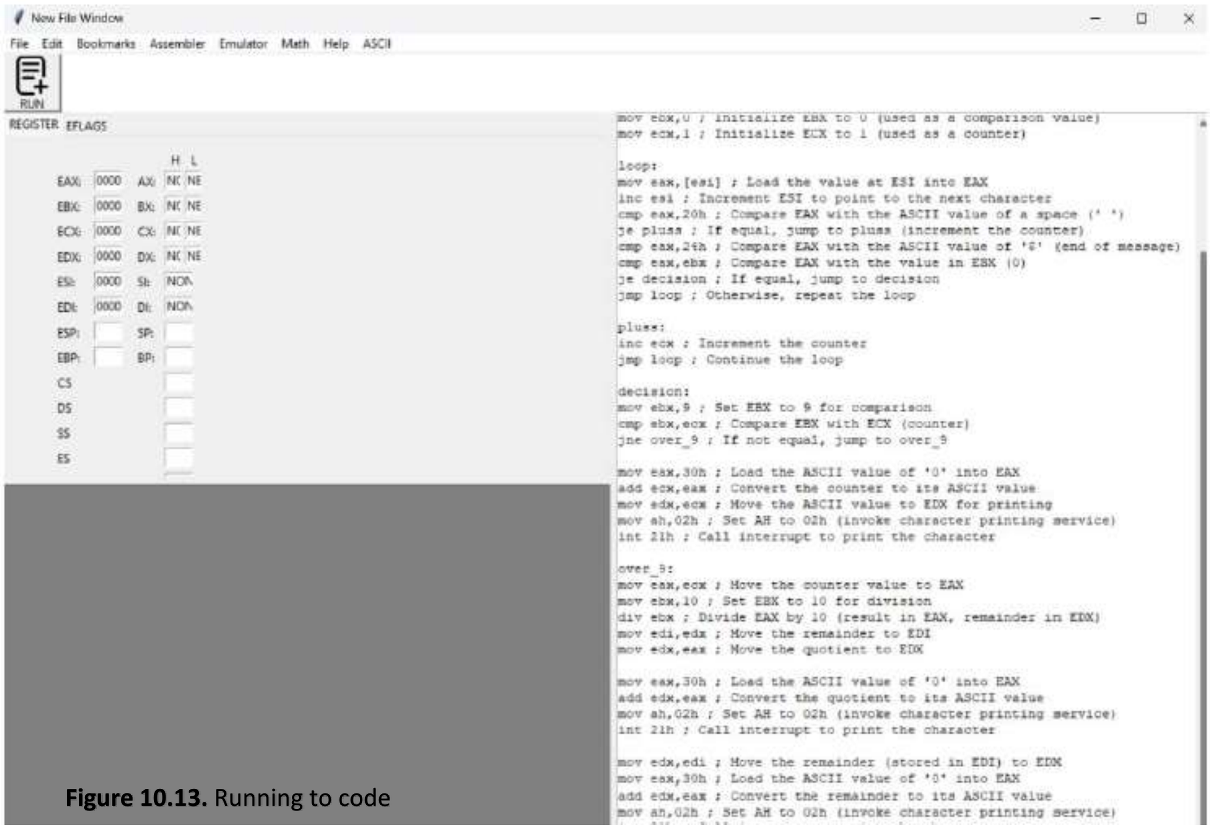
Figure 10.12. Flowchart of Experiment

10.2.2.3. DESCRIPTION OF CODE

If we express what is done in this assembly code in a simple way. The code checks how many times it sees the space character (ASCII code 20h) with the help of a counter (ECX). In order to finish the counting process, it looks for the '$' (ASCII code 24h) that should be added to the end of the string.

In this way, the whitespace is counted to find out how many words are in the sentence.

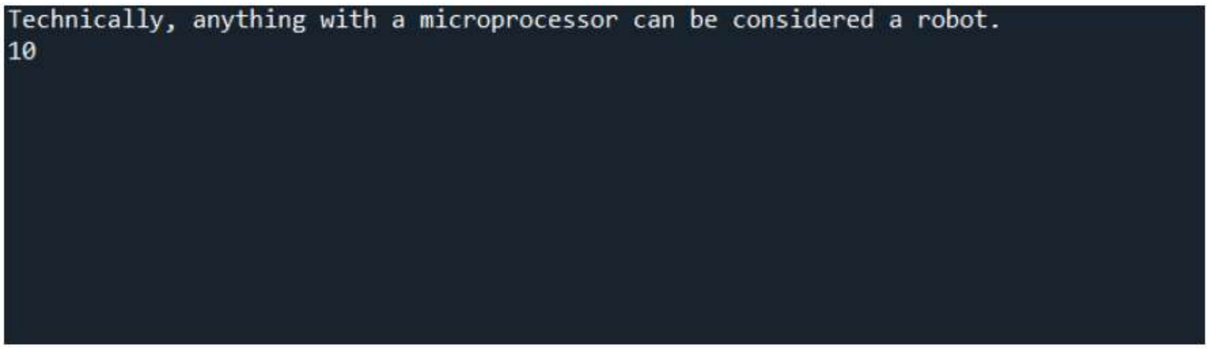
10.2.2.4. PROCESSING OF CODE



```
mov ebx,0 ; Initialize EBX to 0 (used as a comparison value)
mov ecx,1 ; Initialize ECX to 1 (used as a counter)

loop:
mov eax,[esi] ; Load the value at ESI into EAX
inc esi ; Increment ESI to point to the next character
cmp eax,20h ; Compare EAX with the ASCII value of a space (' ')
je pluss ; If equal, jump to pluss (increment the counter)
cmp eax,24h ; Compare EAX with the ASCII value of '$' (end of message)
cmp eax,ebx ; Compare EAX with the value in EBX (0)
je decision ; If equal, jump to decision
jmp loop ; Otherwise, repeat the loop

pluss:
inc ecx ; Increment the counter
jmp loop ; Continue the loop

decision:
mov ebx,9 ; Set EBX to 9 for comparison
cmp ebx,ecx ; Compare EBX with ECX (counter)
jne over_9 ; If not equal, jump to over_9

mov eax,30h ; Load the ASCII value of '0' into EAX
add ecx,eax ; Convert the counter to its ASCII value
mov edx,ecx ; Move the ASCII value to EDX for printing
mov ah,02h ; Set AH to 02h (invoke character printing service)
int 21h ; Call interrupt to print the character

over_9:
mov eax,ecx ; Move the counter value to EAX
mov ebx,10 ; Set EBX to 10 for division
div ebx ; Divide EAX by 10 (result in EAX, remainder in EDX)
mov edi,edx ; Move the remainder to EDI
mov edx,eax ; Move the quotient to EDX

mov eax,30h ; Load the ASCII value of '0' into EAX
add edx,eax ; Convert the quotient to its ASCII value
mov ah,02h ; Set AH to 02h (invoke character printing service)
int 21h ; Call interrupt to print the character

mov edx,edi ; Move the remainder (stored in EDI) to EDX
mov eax,30h ; Load the ASCII value of '0' into EAX
add edx,eax ; Convert the remainder to its ASCII value
mov ah,02h ; Set AH to 02h (invoke character printing service)
```

Figure 10.13. Running to code

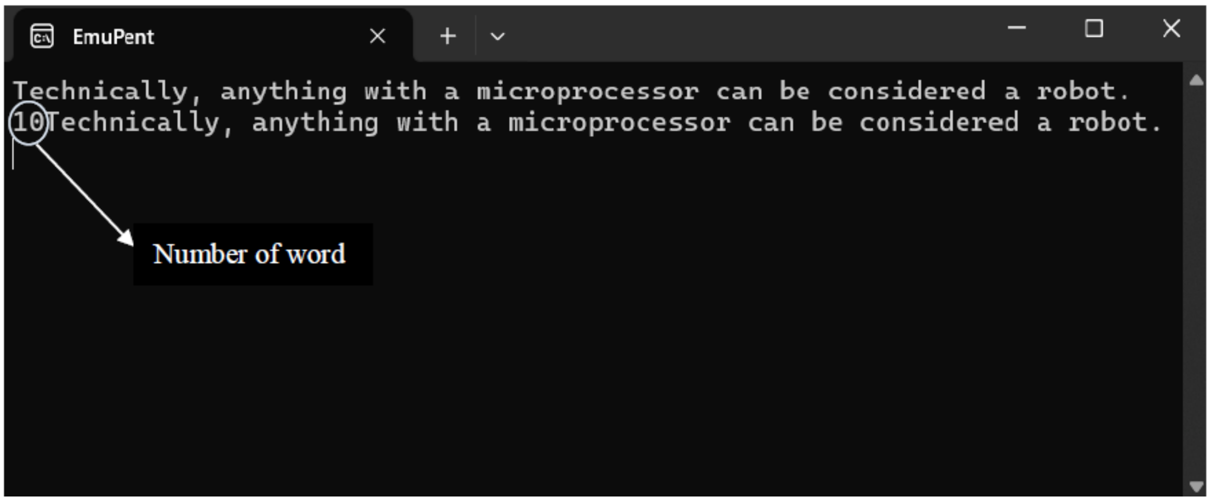
When we run the code in the Spyder environment where we designed EMUPENT, the expected output appears in the python console.



```
Technically, anything with a microprocessor can be considered a robot.
10
```

Figure 10.14. Output of the code in Spyder

We need to run the code once more to display the word count.



```
Technically, anything with a microprocessor can be considered a robot.
10Technically, anything with a microprocessor can be considered a robot.
```

Figure 10.15. Second outout of the code in EMUPENT

10.2.3. PALINDROME EXPERIMENT

We are creating a Pentium emulator using the Python programming language, taking Emu8086 as a reference. Our emulator includes Logic, Arithmetic, and Comparison commands. In addition, we can perform operations such as receiving output from the console and receiving data from the console using int21h. We can also create loops using comparison commands and JMP commands. I n this document, we will explain the commands in our emulator with an example of a palindrome assembly code.

10.2.3.1. CODE

```
.data
msg     db 'Enter a word: $'
msg4 db 'WELCOME TO THE PALINDROME EXPERIMENT $'
msg5 db 'To check whether the letters from the first digit to the median digit of a word are identical to the letters from the last digit to the median digit (If the string is more than 30 letters, you will get an error message.): $'
msg1     db 'It is a Palindrome $'
msg2     db 'It is not a Palindrome $'
msg3     db 'Length is too long $'
buffer  db 10,?, 10 dup('$')
.code
mov edx,offset msg4    ;Load the address of the "WELCOME TO THE PALINDROME EXPERIMENT" message into edx
mov ah,09h                     ;Set AH to 09h (invoke string printing service)
int 21h              ;Call interrupt to print the message

mov edx,offset msg5    ;Load the address of the explanation message for palindrome check into edx
mov ah,09h           ;Set AH to 09h (invoke string printing service)
int 21h            ;Call interrupt to print the message
```

```
ank:
mov edx,offset msg     ;Load the address of the message prompting for user input into edx
mov ah,09h             ;Set AH to 09h (invoke string printing service)
int 21h                ;Call interrupt to print the message

mov edx,offset buffer  ;Load the address of buffer to edx to get user input
mov ah,0Ah                   ;Set AH to 0Ah (invoke buffered input service)
int 21h                ;Call interrupt to get input from the user

mov edi,esi            ;Copy the address of input from ESI to EDI
mov eax,[buffer]       ;Get the length of input stored in buffer
dec eax                ;Because an extra byte is stored in the buffer structure after the 0Ah call
add edi,eax            ;Set EDI to point to the end of input
mov ebx,30             ;Load 30 which is the MAXIMUM Length into EBX
cmp ebx,eax            ;Check if the length of input is less than 30
jl print_msg3          ;If it's less than 30, jump to print_msg3
check_palindrome:
mov eax,[esi]          ;Get a character from the start address
mov ebx,[edi]          ;Get a character from the end address
cmp eax,ebx            ;Compare the characters
jne print_message2     ;If they are different, jump to print_message2
inc esi                ;Move the start address to the next character
dec edi                ;Move the end address to the previous character
cmp edi,esi            ;Check if there is an intersection between EDI and ESI
jl check_palindrome    ;If there's no intersection, continue palindrome check

print_message:         ;Print msg1 which is 'It is a Palindrome'
mov edx,offset msg1
mov ah,09h
int 21h
jmp ank                ;Jump to ank

print_message2:        ;Print msg2 which is 'It is not a Palindrome'
mov edx,offset msg2
mov ah,09h
int 21h
jmp ank                ;Jump to ank

print_msg3:            ;Print msg3 which is 'Length is too long'
mov edx,offset msg3
mov ah,09h
int 21h
jmp exit               ;Jump to ank

exit:
```

10.2.3.2. FLOWCHART OF CODE

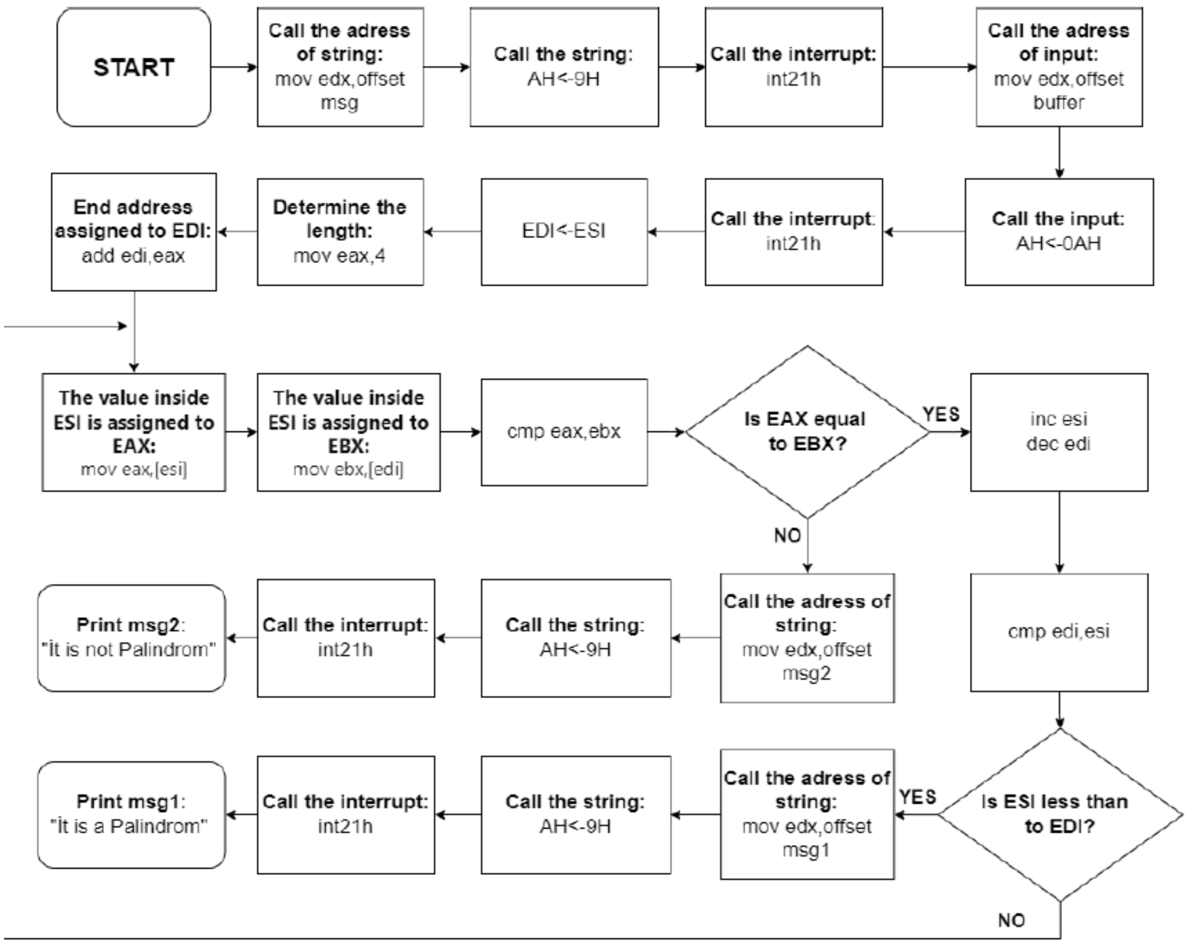

START
Call the adress of string: mov edx,offset msg
Call the string: AH<-9H
Call the interrupt: int21h
Call the adress of input: mov edx,offset buffer
Call the input: AH<-0AH
Call the interrupt: int21h
EDI<-ESI
Determine the length: mov eax,4
End address assigned to EDI: add edi,eax
The value inside ESI is assigned to EAX: mov eax,[esi]
The value inside ESI is assigned to EBX: mov ebx,[edi]
cmp eax,ebx
Is EAX equal to EBX?
YES
inc esi dec edi
NO
Call the adress of string: mov edx,offset msg2
Call the string: AH<-9H
Call the interrupt: int21h
Print msg2: "İt is not Palindrom"
cmp edi,esi
Is ESI less than to EDI?
YES
Call the adress of string: mov edx,offset msg1
Call the string: AH<-9H
Call the interrupt: int21h
Print msg1: "İt is a Palindrom"
NO

Figure 10.16. Flowchart of Code

10.2.3.3. DESCRIPTION OF CODE

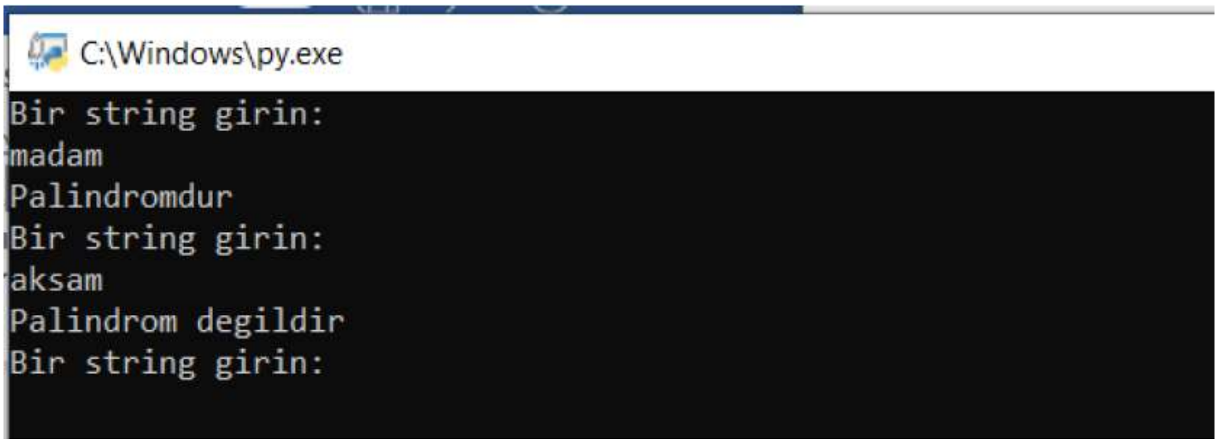
.data
The .data directive usually indicates the section where variables are defined.

msg db 'Enter a word: $'
msg4 db 'WELCOME TO THE PALINDROME EXPERIMENT $'
msg5 db 'To check whether the letters from the first digit to the median digit of a word are identical to the letters from the last digit to the median digit (If the string is more than 30 letters, you will get an error message.): $'
msg1 db 'It is a Palindrom $'
msg2 db 'It is not a Palindrom $'
msg3 db 'Length is too long $'
db→The db directive defines a specific byte value and stores this byte value in memory.
buffer db 10,?, 10 dup('$')

db→ The db directive defines a specific byte value and stores this byte value in memory.

10,?, 10 dup('$')→ It gives an error if these commands are not present, but they currently have no effect on the code.

.code
We need to write this directive to be able to write the commands.

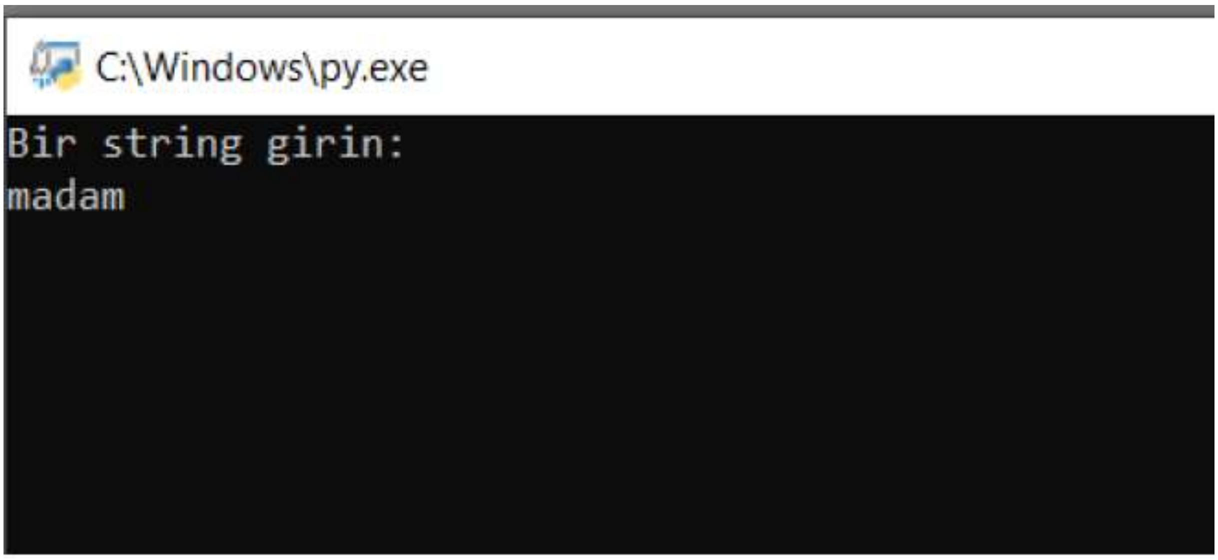


```
C:\Windows\py.exe
Bir string girin:
madam
Palindromdur
Bir string girin:
aksam
Palindrom degildir
Bir string girin:
```

Figure 10.17. Start Loop

Receiving output from the console with int21h:
mov edx,offset msg→It transfers the address of the msg variable in memory to the edx register.
mov ah,09h→ We need to assign the value 9 to ah to get the output of the string.
int 21h→ **IT calls the int21h interrupt.**

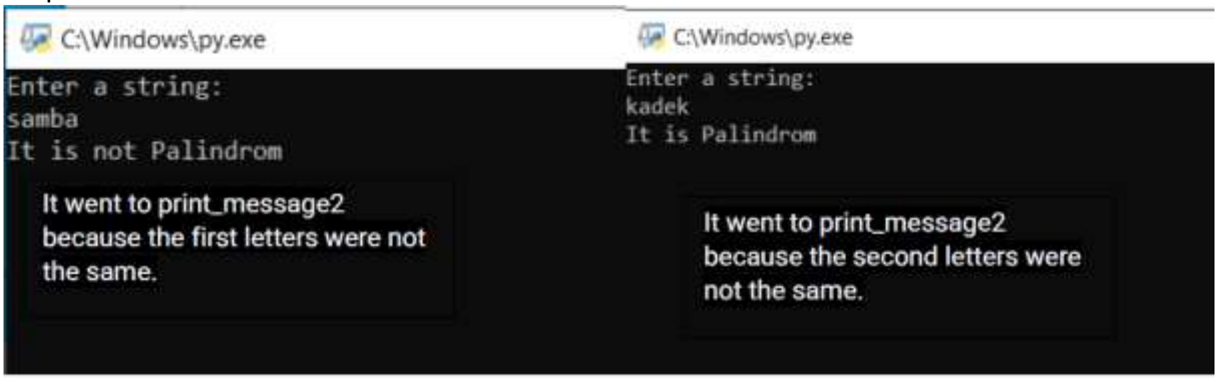
Receiving input from the console with int21h:
mov edx,offset msg→It transfers the address of the msg variable in memory to the edx register.
mov ah,0Ah → We need to assign the value A to ah to get the input of the string.
int 21h→ It calls the int21h interrupt.



```
C:\Windows\py.exe
Bir string girin:
madam
```

Figure 10.18. Output and Input with int21h

`mov edi,esi`→ We assign the esi value, which is the starting address of the string, to edi. Here, the esi address is initially assigned to 9. We can determine it ourselves with the MOV command.
`mov eax,[buffer]`→ It assigns the length of the word we write to the eax register.
`dec eax` →We reduce the length by 1 because there is also a value at the zeroth index
So, if the value we initially equate to esi is 9, we do 9+4 and set the end address as 13. This means that the ASCII codes of each character of the entered 5-digit string will be assigned to these addresses: 9, 10, 11, 12, 13.
`add edi,eax`→ We are determining the end address by assigning the value in eax to edi.
`mov ebx,20`→ We determine the maximum length
`cmp eax,ebx`→ We compare eax and ebx to check whether they exceed the length limit.
`jl print_msg3` → If the number of string digits entered is large, it goes to the print_msg3 loop.
`check_palindrome:`→ The start of the loop created for palindrome control.
`mov eax,[esi]`
`mov ebx,[edi]`
It assigns the ASCII code at the address of ESI to EAX, and the ASCII code at the address of EDI to EBX.
`cmp eax,ebx`→ It compares the ASCII values assigned to EAX and EBX at these addresses. If they are the same, it means the same letter, if not, it means a different letter.
jne print_message2→ If the two ASCII codes are not equal, it goes to the print_message2 loop.



```
Enter a string:
samba
It is not Palindrom
```

It went to print_message2 because the first letters were not the same.

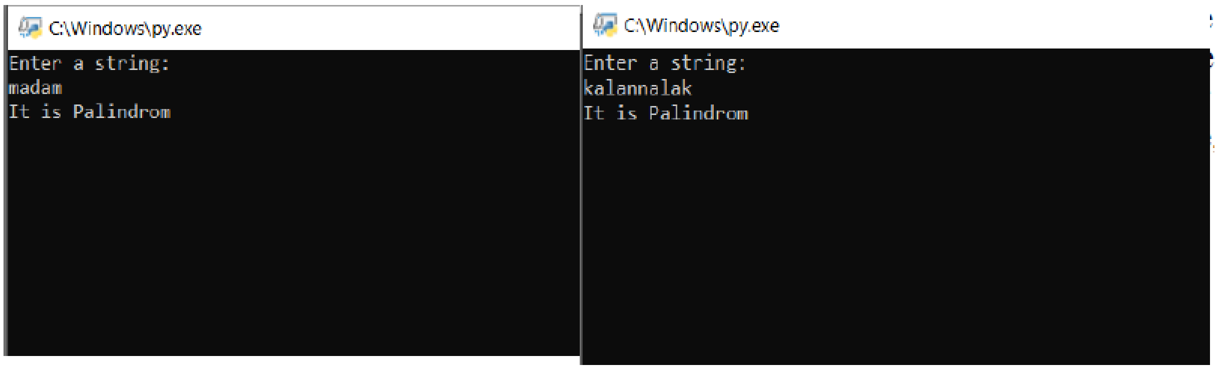
```
Enter a string:
kadek
It is Palindrom
```

It went to print_message2 because the second letters were not the same.

Figure 10.19. print_message2

`inc esi`
`dec edi`
We increase the start address by 1 and decrease the end address by 1. This way, we can check the letters one by one in order.
cmp edi,esi → We compare the address values of esi and edi.

`jl check_palindrome` → If ESI is less than EDI, it goes back to the check_palindrome loop. It looks again at the ASCII code in the increased ESI value, and looks again at the ASCII code in the decreased EDI value, and compares them. If they are different, it goes to print_message2. If they are the same, it performs this loop until the ESI value is greater tha the EDI value. When it is greater, it does not enter the loop and continues with the code, giving the message ‘It is a palindrome’



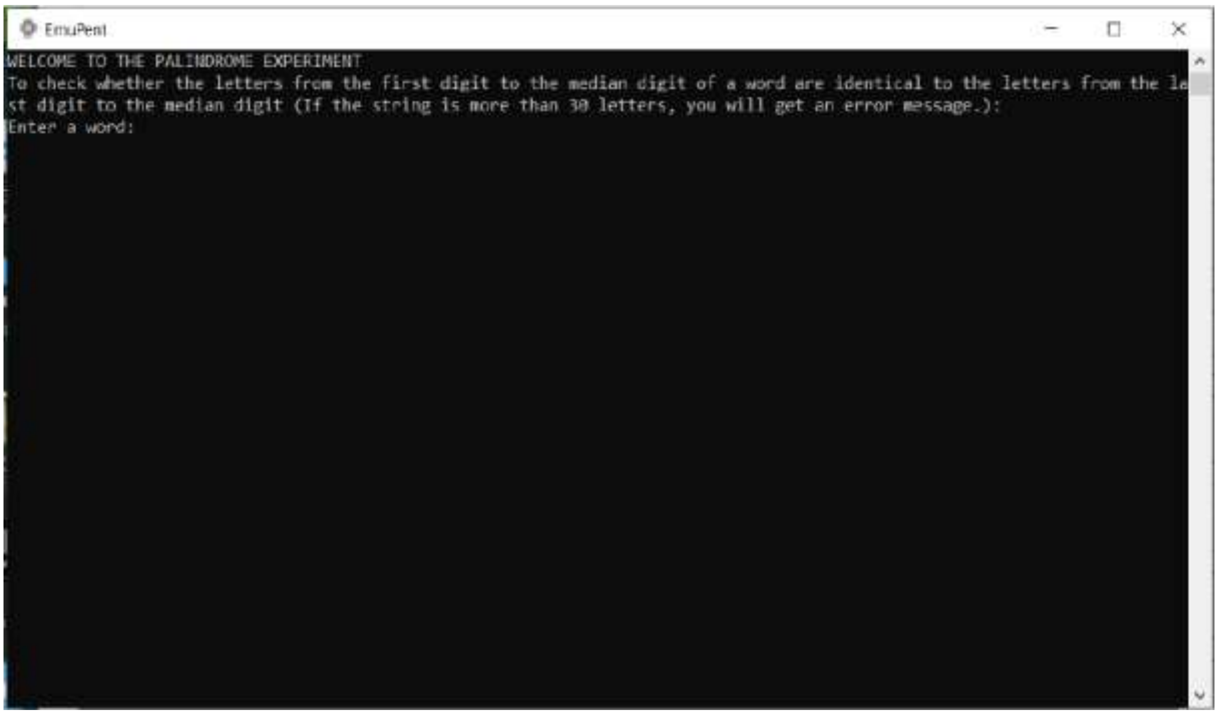
```
C:\Windows\py.exe
Enter a string:
madam
It is Palindrom
```

```
C:\Windows\py.exe
Enter a string:
kalannalak
It is Palindrom
```

Figure 10.20. print_message1

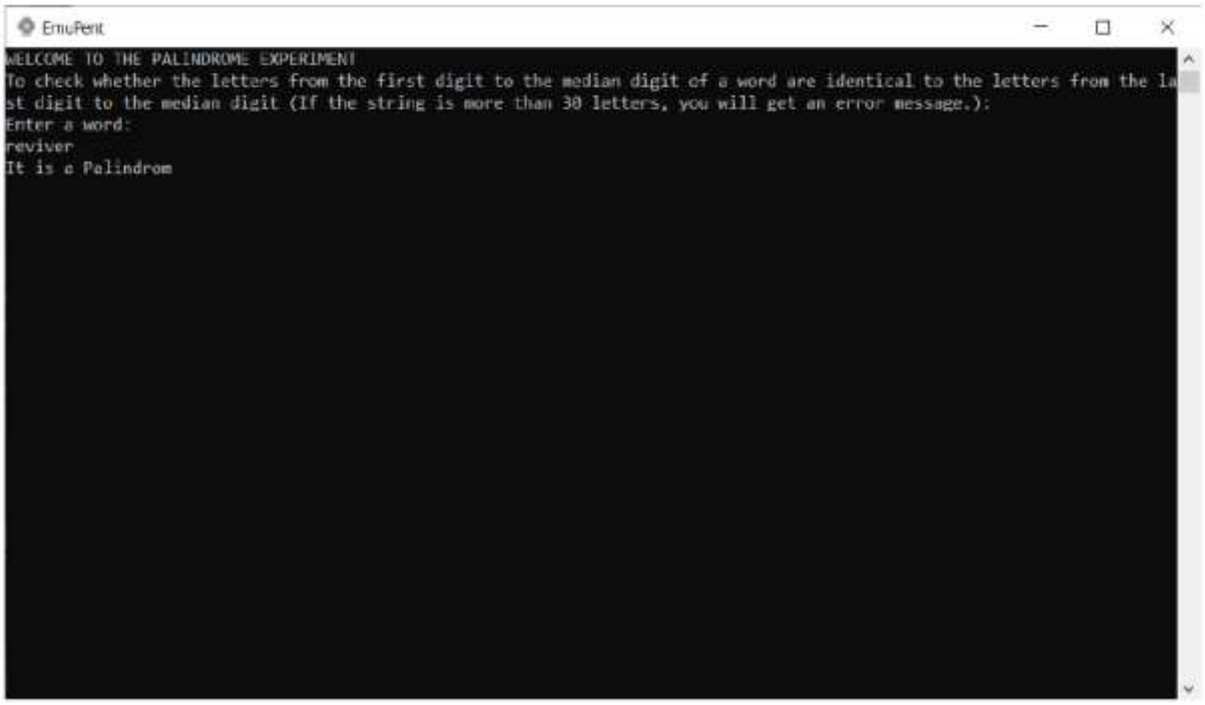
print_message:
mov edx,offset msg1
mov ah,09h
int 21h
jmp finish
print_message2:
mov edx,offset msg2
mov ah,09h
int 21h
jmp finish
print_msg3:
mov edx,offset msg3
mov ah,09h
int 21h
jmp finish
finish: → We created an empty loop to finish the code

10.2.3.4. PROCESSING OF CODE



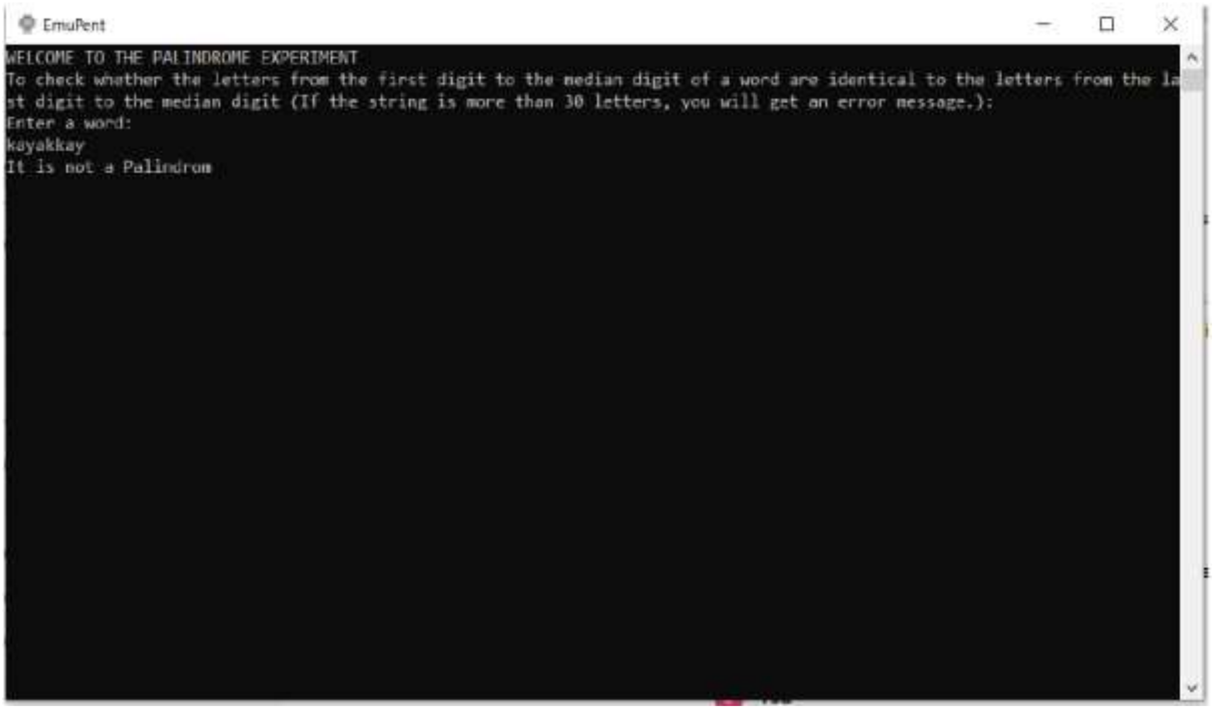
```
EmuPent
WELCOME TO THE PALINDROME EXPERIMENT
To check whether the letters from the first digit to the median digit of a word are identical to the letters from the la
st digit to the median digit (If the string is more than 30 letters, you will get an error message.):
Enter a word:
```

Figure 10.21. Output of Console screen.



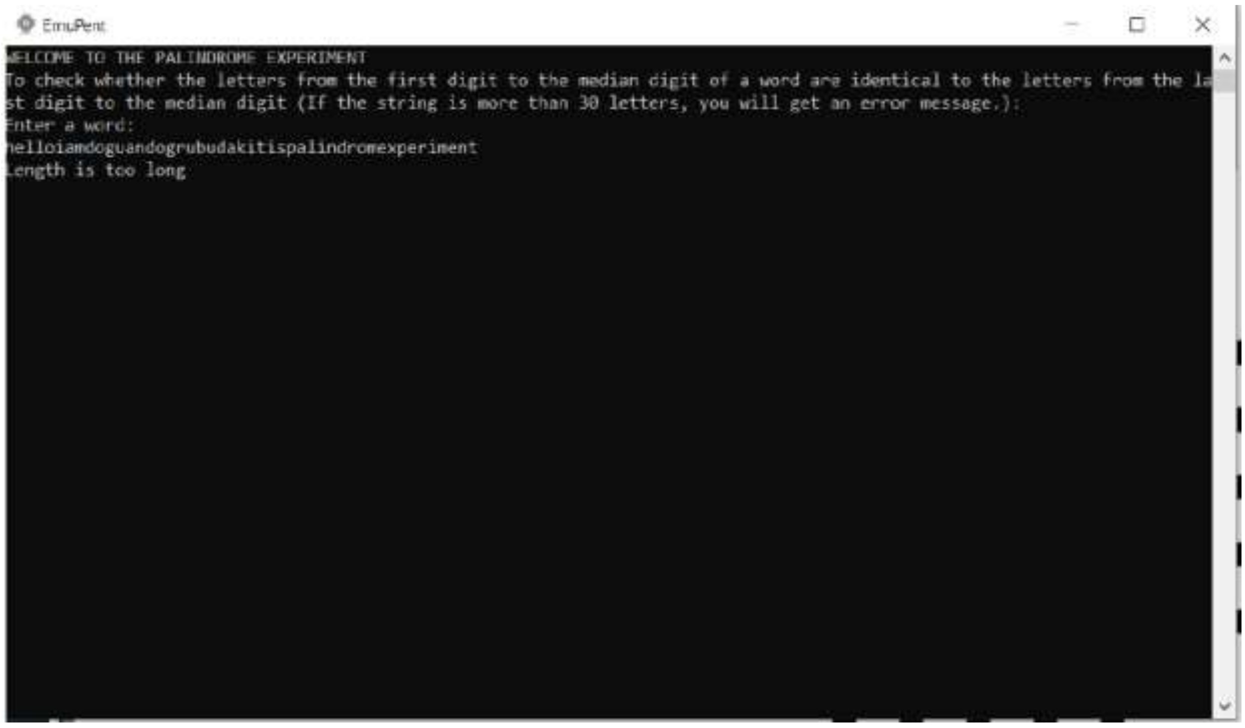
```
WELCOME TO THE PALINDROME EXPERIMENT
To check whether the letters from the first digit to the median digit of a word are identical to the letters from the la
st digit to the median digit (If the string is more than 30 letters, you will get an error message.):
Enter a word:
reviver
It is a Palindrom
```

Figure 10.22. Palindrom output



```
WELCOME TO THE PALINDROME EXPERIMENT
To check whether the letters from the first digit to the median digit of a word are identical to the letters from the la
st digit to the median digit (If the string is more than 30 letters, you will get an error message.):
Enter a word:
kayakkay
It is not a Palindrom
```

Figure 10.23. Not Palindrom output



```
WELCOME TO THE PALINDROME EXPERIMENT
To check whether the letters from the first digit to the median digit of a word are identical to the letters from the la
st digit to the median digit (If the string is more than 30 letters, you will get an error message.):
Enter a word:
helloiamdoguandogrubudakitispalindromexperiment
Length is too long
```

Figure 10.24 Long Length

10.2.4. MOUSE INTERRUPT EXPERIMENT

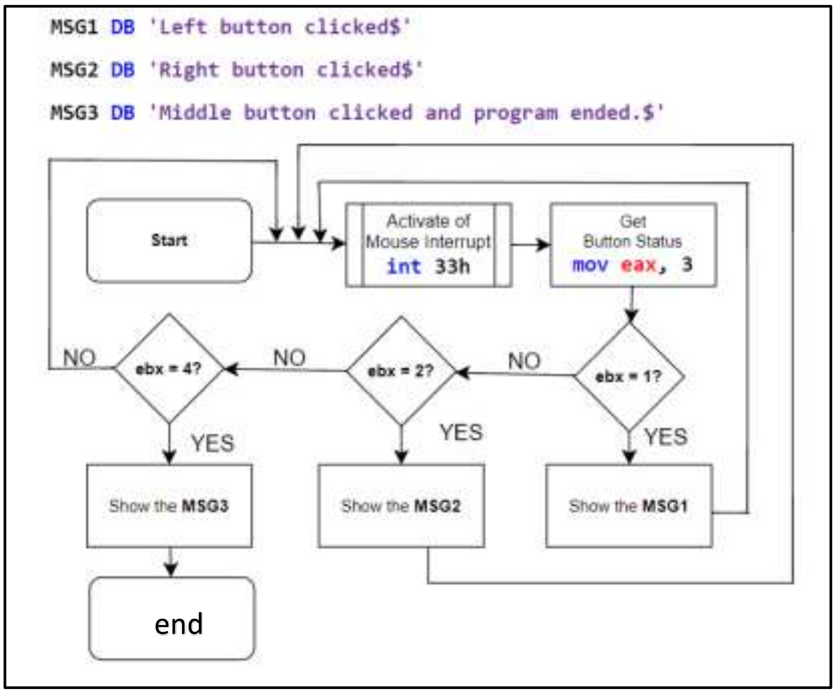
In this experiment, we understand the right click, left click, and middle cursor of the mouse and print this on the console.

10.2.4.1. CODE

```
.DATA
MSG1 DB 'Left button clicked$'
MSG2 DB 'Right button clicked$'
MSG3 DB 'Middle button clicked and program ended.$'
.CODE
main_loop:
; Check mouse button status
mov eax, 3
int 33h
; Compare for left button click
mov ecx,1
cmp ebx,ecx
jne check_right
je go_left
jmp main_loop
; Compare for right button click
check_right:
mov ecx,2
cmp ebx,ecx
jne check_middle
```

```
je go_right
; Compare for middle button
check_middle:
mov ecx,4
cmp ebx,ecx
jne main_loop
je go_middle
; Display middle button message
go_middle:
mov edx,offset MSG3
mov ah, 09h
int 21h
jmp end
; Display right button message
go_right:
mov edx,offset MSG2
mov ah, 09h
int 21h
jmp main_loop
; Display left button message
go_left:
mov edx,offset MSG1
mov ah, 09h
int 21h
jmp main_loop
; End program
end:
```

10.2.4.2. FLOWCHART OF CODE



```
MSG1 DB 'Left button clicked$'
MSG2 DB 'Right button clicked$'
MSG3 DB 'Middle button clicked and program ended.$'
```

Figure 10.25. Flowchart of Mouse İnterrupt

10.2.4.3. DESCRIPTION OF CODE

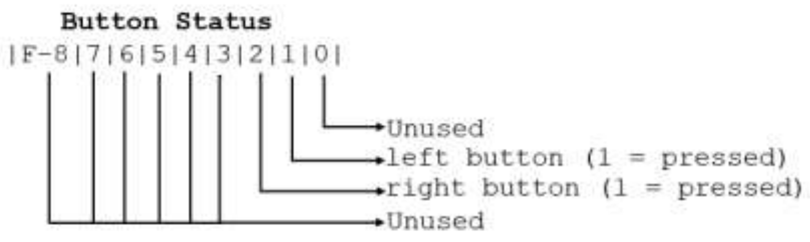
```
.DATA
MSG1 DB 'Left button clicked$'
MSG2 DB 'Right button clicked$'
MSG3 DB 'Middle button clicked and program ended.$'
```

There are strings here that will be printed on the console screen each time the mouse is clicked.

```
.CODE
main_loop:
; Check mouse button status
mov eax, 3
int 33h
```

INT 33,3 - Get Mouse Position and Button Status

```
AX = 03
      on return:
      CX = horizontal (X) position  (0..639)
      DX = vertical (Y) position  (0..199)
      BX = button status:
```



```
- values returned in CX, DX are the same regardless of video
mode
```

```
mov ecx,1
cmp ebx,ecx
jne check_right
je go_left
jmp main_loop
```

If EBX equals 2, the left button interrupt is activated and MSG2 is printed to the console; otherwise, it goes to the check_right loop.

```
check_right:
; Compare for right button click
mov ecx,2
cmp ebx,ecx
jne check_middle
je go_right
```

If EBX equals 4, the left button interrupt is activated and MSG3 is printed to the console; otherwise, it goes to the main loop.

```
check_middle:
  ; Compare for right button click
mov ecx,4
cmp ebx,ecx
jne main_loop
je go_middle
```

Display cursor button message

```
go_middle:
; Display cursor button message
mov edx,offset MSG3
mov ah, 09h
int 21h
jmp end
```

Display right button message

```
go_right:
; Display right button message
mov edx,offset MSG2
mov ah, 09h
int 21h
jmp main_loop
```

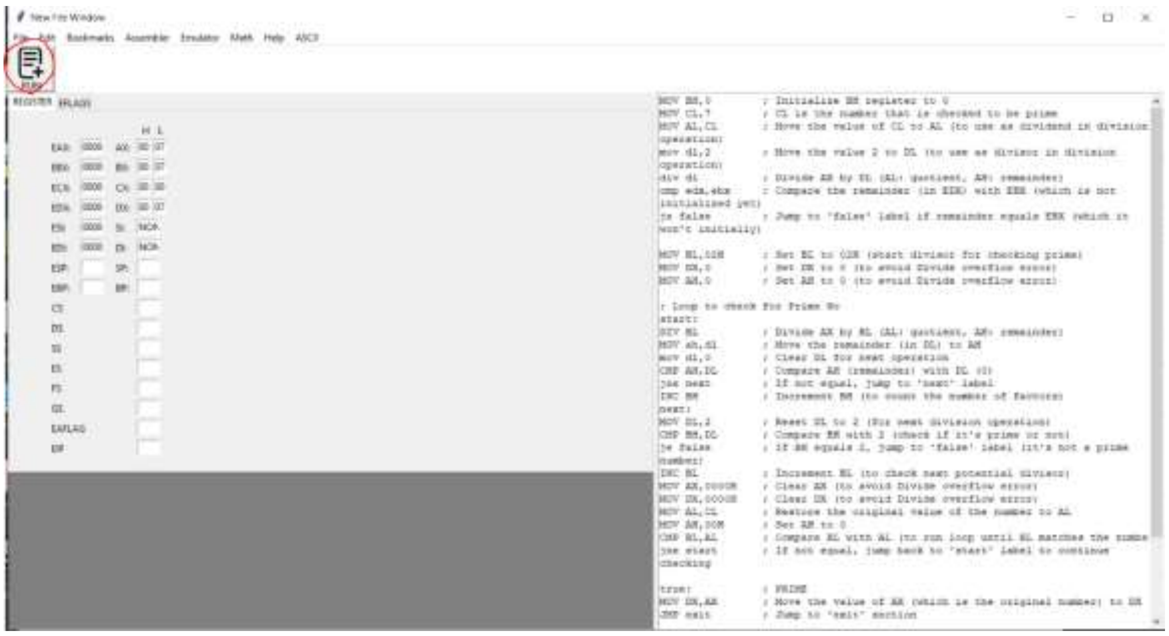
Display left button message.

```
go_left:     ; Display left button message
mov edx,offset MSG1
mov ah, 09h
int 21h
jmp main_loop
```

End program

```
end:
    ; End program
```

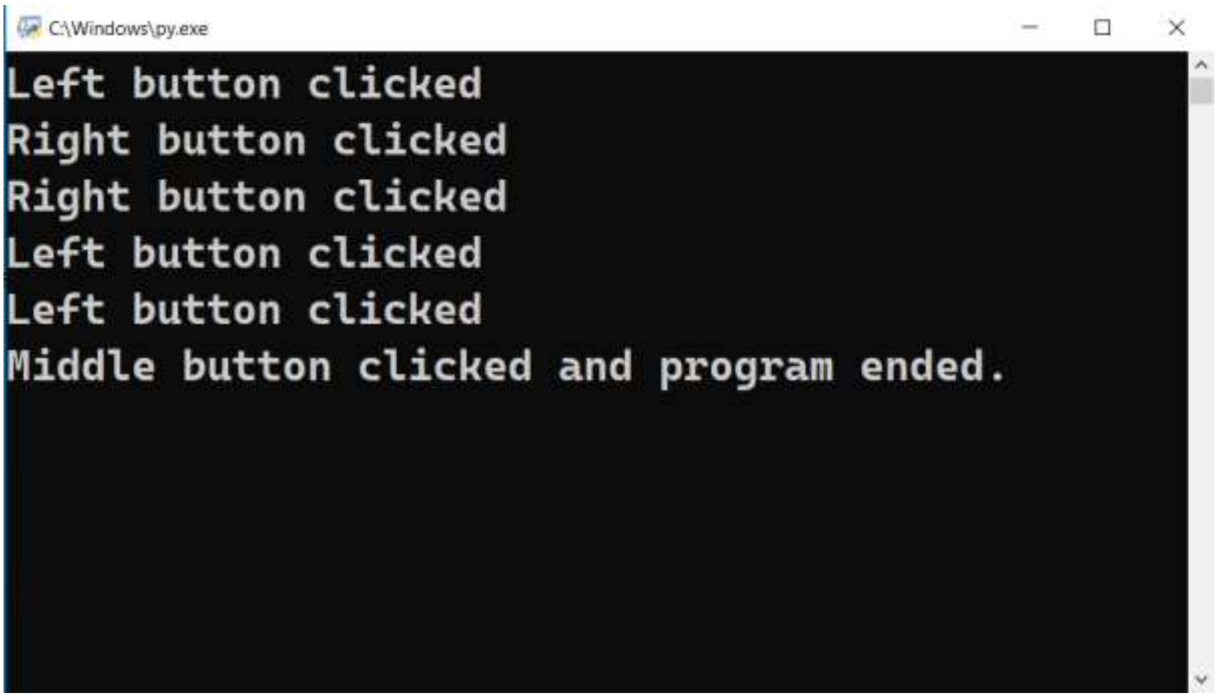
10.2.4.4. PROCESSING OF CODE



```
MOV BH,0          ; Initialize BH register to 0
MOV CL,7          ; CL is the number that is checked to be prime
MOV AL,CL         ; Move the value of CL to AL (to use as dividend in division
operation)
mov dl,2          ; Move the value 2 to DL (to use as divisor in division
operation)
div dl            ; Divide AX by DL (AL: quotient, AH: remainder)
cmp edx,ebx       ; Compare the remainder (in EDX) with EBX (which is not
initialized yet)
je false          ; Jump to 'false' label if remainder equals EBX (which it
won't initially)

MOV BL,02H        ; Set BL to 02H (start divisor for checking prime)
MOV DX,0          ; Set DX to 0 (to avoid Divide overflow error)
MOV AH,0          ; Set AH to 0 (to avoid Divide overflow error)

; Loop to check for Prime No
start:
DIV BL            ; Divide AX by BL (AL: quotient, AH: remainder)
MOV ah,dl         ; Move the remainder (in DL) to AH
mov dl,0          ; Clear DL for next operation
CMP AH,DL         ; Compare AH (remainder) with DL (0)
jne next          ; If not equal, jump to 'next' label
INC BH            ; Increment BH (to count the number of factors)
next:
MOV DL,2          ; Reset DL to 2 (for next division operation)
CMP BH,DL         ; Compare BH with 2 (check if it's prime or not)
je false          ; If BH equals 2, jump to 'false' label (it's not a prime
number)
INC BL            ; Increment BL (to check next potential divisor)
MOV AX,0000H      ; Clear AX (to avoid Divide overflow error)
MOV DX,0000H      ; Clear DX (to avoid Divide overflow error)
MOV AL,CL         ; Restore the original value of the number to AL
MOV AH,00H        ; Set AH to 0
CMP BL,AL         ; Compare BL with AL (to run loop until BL matches the numbe
jne start         ; If not equal, jump back to 'start' label to continue
checking

true:             ; PRIME
MOV DX,AX         ; Move the value of AX (which is the original number) to DX
JMP exit          ; Jump to 'exit' section
```

Figure 10.27 Mouse İnterrupt assembly code

When we click the left button on the mouse, it writes ‘‘Left button clicked’’ on the console, when we click the right button, it writes ‘‘Right button clicked’’, and when we click the middle cursor, it writes ‘‘Middle button clicked, and program ended.’’ and the program closes. In other cases, the program enters a loop and returns to the beginning.



C:\Windows\py.exe

```
Left button clicked
Right button clicked
Right button clicked
Left button clicked
Left button clicked
Middle button clicked and program ended.
```

Figure 10.28 Output of Mouse Interrupt

10.3. ASSEMBLY CODES OF EXPERIMENTS

We uploaded the assembly code I created in this code to the links below. You can download the .asm file there and open this file from the entry interface by doing ‘Open File’.

Prime Number: https://raw.githubusercontent.com/Ozgur-Tanriverdi/EMUPENT/main/ASM_Code_Prime_Number/PRIME_example.asm

Counting Word: https://raw.githubusercontent.com/Ozgur-Tanriverdi/EMUPENT/main/ASM_Code_Counting_Number_of_Word/Number_of_Word.asm

Palindrome: https://raw.githubusercontent.com/Ozgur-Tanriverdi/EMUPENT/main/ASM_Code_Palindrom/Palindrome_example.asm

Mouse Operation: https://raw.githubusercontent.com/Ozgur-Tanriverdi/EMUPENT/main/ASM_code_Mouse_Operations/EXP4_Mouseinterrupt.asm

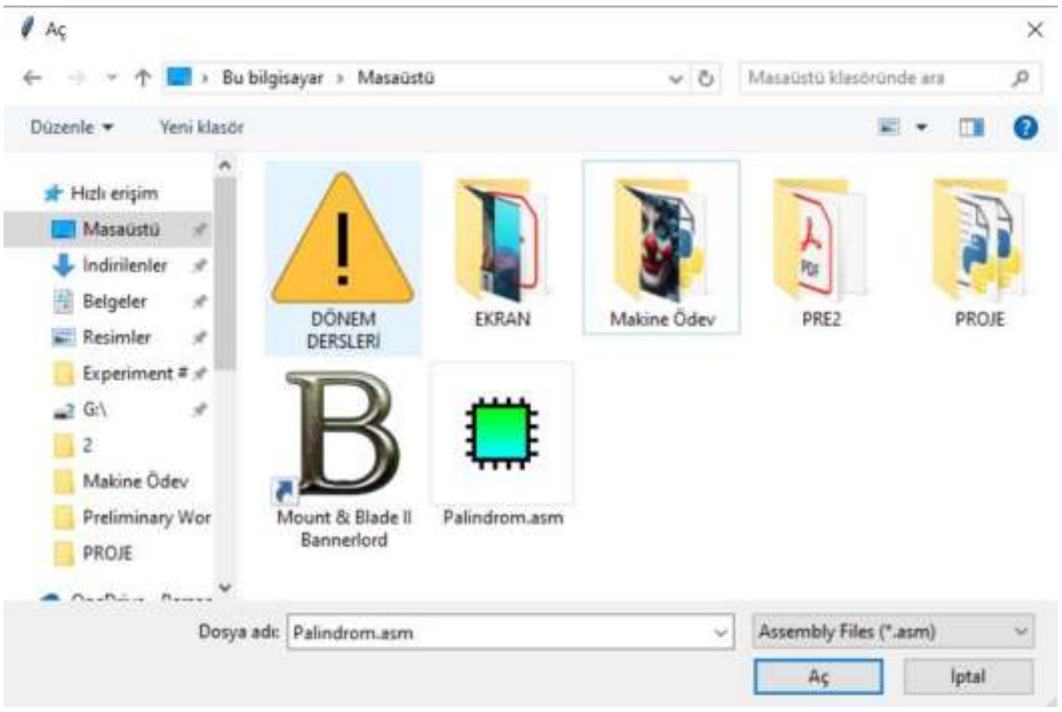


Figure 10.29. Open File window

10.4. MEETING THE CONSTRAINTS AND ENGINEERING STANDARDS

10.4.1. ENGINEERING STANDARDS

10.4.1.1 ISO/IEC/IEEE 12207:2017

Planning and requirement analysis, defining the requirements, designing the product architecture, creating or developing the product, testing the product, and all stages of evaluation have been carried out.

10.4.1.2. ISO 31-11

Here, it is compatible because we use 'mathematical operations' and 'mathematical logic'.

10.2.1.3 IEEE 830

The IEEE 830 standard specifies how software requirements should be documented. Applying this standard in our project ensures that all functional and non-functional requirements of the Pentium emulator are captured systematically and comprehensively. It is compatible because ours is a software project and we have prepared plenty of reports during the project process.

10.2.1.4. ISO/IEC 9126 - Software Product Quality

For our Pentium emulator, adhering to ISO/IEC 9126 ensures that we systematically evaluate and improve its functionality, reliability, usability, efficiency, maintainability, and portability.

10.4.2. SUSTAINABLE DEVELOPMENT GOALS

The emulator is still suitable for goal 4, goal 11, goal 12, goal 17, but not the goal 9 we mentioned before:

Goal 4: Quality education
The emulator we have designed provides the possibility to test the Pentium directly. This can be a valuable resource for students and educators in computer science and related fields. Emulatörümüzü kullanarak bazı deneyler öğrenciler tarafından denenmiştir.

Goal 9: Industry, Innovation and Infrastructure
Understanding and emulating older architectures is important for preserving digital heritage.

Goal 12: Responsible consumption and production
Developing an emulator can reduce the need for physical hardware and contribute to more sustainable practices in computing.

Goal 17: Partnerships towards the goals
Developing a Pentium emulator can inspire collaboration and partnerships within the technology community. This allows developers, educators and enthusiasts to come

together and share knowledge and contribute to common goals in the field of computer science.

10.5. COST ANALYSIS

Since it is a software project, there has been no cost. We have paid for some courses on the internet where we learned some topics.

11. TEAMWORK

As team members, we worked intensively and harmoniously as a good team. While there are many things that both of us have done together, Doğukan Doğrubudak has worked more on issues such as interface design and general layout, and Özgür Tanrıverdi has worked more on issues such as processing the necessary functions for the Emulator.

Since jump instructions move from one line to another line, in order to avoid intermediate operations and infinite loops, Özgür created line indexes and defined these line indexes correctly in the functions where jump instructions are used. Thus, Özgür has made the function that runs in the background of the program. Doğukan has performed the operations of processing these commands into the text editor.

At the end of the project, generally, most parts were worked on together.

12.COMMENTS AND CONCLUSIONS

In conclusion, we have managed to complete our graduation project despite some shortcomings. Our emulator can perform interrupt operations, string operations, flag operations, and mouse operations with approximately 30 commands. In addition, we were able to produce a product that is suitable for the purpose of the project. Although we couldn't add all the features of the Pentium, we were able to conduct 4 experiments. We couldn't fully realize the 4th experiment, but we created a code that can use the mouse interrupt, which is its main purpose, and put it as the 4th experiment. Along with these, we had difficulties in issues such as creating memory, creating a console, integrating general functions into the entire program. As a result, we were able to conduct 4 experiments. When we look at it in terms of competence, it doesn't seem very sufficient because we created the emulator according to these four experiments, not a complete 1 emulator according to the Pentium processor. However, due to the ease and understandability of the codes, it became an emulator that can be developed with an open end.

13.REFERENCES

[1] M. A. Mazidi and J. G. Mazidi, The 80X86 IBM PC and Compatible Computers: Assembly Language, Design, and Interfacing. 1997. doi: 10.1604/9780137585090

[2] M. Dönmez, "Mikroişlemci Nasıl Çalışır? | muratdonmez.com.tr," muratdonmez.com.tr | Akıllı ev sistemleri, IOT, ESP8266, NodeMCU, Raspberry ve Elektronik ürün incelemeleri, proje örnekleri, Oct. 20, 2020. https://www.muratdonmez.com.tr/mikroislemci-nasil-calisir/#google_vignette

[3] "ISO/IEC/IEEE 12207:2017," ISO, Feb. 04, 2021. https://www.iso.org/standard/63712.html

[4] https://www.undp.org/sustainable-development-goals

[5] "SDLC - Overview," SDLC - Overview. https://www.tutorialspoint.com/sdlc/sdlc_overview.htm

[6] "Python GUIs — Create GUI applications with Python and Qt," Python GUIs, Jan. 18,

[7] 2023. https://www.pythonguis.com//

F. Prasetya, D. T. Saifuddin, and N. Y. Ansir, "https://www.ijser.org/research-paper- publishing-april-2020.aspx," International Journal of Scientific & Engineering Research, vol. 11, no. 04, pp. 1801–1806, Apr. 2020, doi: 10.14299/ijser.2020.04.06.