Final Project

ECE 3104: Microprocessor & Microcomputer Laboratory

Designing a Simple Calculator with 8086 Microprocessor

Group Members:

Nishat Yasmin Anika, Roll: 2009056

Pritam Kumar Das, Roll: 2009057

Md. Polash Ahmed, Roll: 2009058

Md. Habibur Rahman, Roll: 2009059

Md. Akramuddoula, Roll: 2009060

Department of Electronics and Communication Engineering

Khulna University of Engineering and Technology

Khulna-9203, Bangladesh

May 14, 2024

Objective:

- To design a simple calculator to perform mathematical operations: Addition, Subtraction, Multiplication and Division of two 8-bit operands in the range of 0h-9h with 8086 microprocessor and display result.

Introduction:

8086 is a 16-bit processor. Its ALU, internal registers work with 16-bit binary words. It has a 16-bit data bus to read or write data. Address bus size is 20-bit which means it can address up to 220=1 MB memory location. The frequency range of the 8086 is 6-10 MHz Like 8085, 8086 too can do only fixed-point arithmetic as the integrated circuit technology of that time did not permit to put additional circuitry on 8006 to do floating-point operations. Intel had designed the coprocessor 8087 that can do floating-point arithmetic and other complex mathematical operations. The 8086 can work in conjunction with 8087 to do both fixed-point, floating point and other complex mathematical functions. The 8086 is designed to operate in two modes, minimum and maximum mode. In the minimum mode, the 8086 processor works in a single processor environment and generates control bus signals. The maximum mode is designed to be used work with coprocessor 8087. The 8086 works in a multiprocessor environment. Control signals for memory and I/O are generated by an external BUS controller. It can pre-fetch up to six instruction bytes from memory and queues them in order to speed up instruction execution. It requires 5V power supply and uses a 40-pin dual in line package. 8086 has two blocks- BIU and EU. The BIU performs all bus operations such as instruction fetching, reading and writing operands for memory and calculating addresses of the memory operands, prefetch of up to six bytes of instruction code. The instruction bytes are transferred to the instruction queue. The EU executes instructions from the instruction system byte queue.

Theory:

A simple calculator implemented using the 8086-assembly language typically involves taking input from the user, performing arithmetic operations (such as addition, subtraction, multiplication, and division), and displaying the results. Here's a general introduction of how we could implement such a calculator:

<u>Display Interface:</u> You can start by displaying a menu to the user, prompting them to choose an operation (e.g., addition, subtraction, multiplication, division). This can be done by outputting strings to the console using interrupts like int 21h.

<u>Input Handling:</u> After displaying the menu, wait for user input. You can use interrupts like int 16h to read keypresses from the keyboard. Convert the user's input into a format suitable for processing (e.g., ASCII to numeric values).

Arithmetic Operations: Based on the user's choice, perform the corresponding arithmetic operation. You can implement algorithms for addition, subtraction, multiplication, and division in assembly language. Remember to handle special cases like division by zero.

Display Result: After performing the calculation, display the result to the user. Convert the numeric result into a human-readable format (e.g., ASCII) and output it to the console.

Repeat or Exit: After displaying the result, give the user the option to perform another calculation or exit the program. Implement logic to loop back to the beginning or exit gracefully based on user input.

Error Handling: Handle error cases such as invalid input or division by zero. Display appropriate error messages to the user and handle these cases gracefully.

Software:

EMU8086 - MICROPROCESSOR EMULATOR

Features:

1. It can take input operands from user.
2. It can perform operation asking which operator to be used in runtime.
3. Invalid inputs are rejected showing ‘Error choice’ message.
4. Display a message in case of divisor is taken zero while executing divide operation.

Assembly Language:

```
    .model small
.stack 100h


.data
operand1 db ?
operand2 db ?
result db ?
operator db ?
msg0 DB 10,13,"-------------Wellcome to 8086 calculator-------------- $"
msg3 DB 10,13,10,13, "Enter the operator (+, -, *, /): $"
```

```
msg1 DB 10,13,10,13, "Enter the first operand: $"
msg2 DB 10,13,10,13, "Enter the second operand: $"
msg5 DB 10,13,10,13, "The Result is:$"
msg_divide_by_zero db DB 10,13, "I can't divide by zero. I Quit.$"
msg_invalid_operator db "Invalid operator", 0Dh, 0Ah, "$"
negative_sign db "-"
ascii DB 2 DUP(?)

.code
main proc
   mov ax, @data
   mov ds, ax

   MOV AH, 09h
   LEA DX, msg0
   INT 21h

   ; Input first operand
   MOV AH, 09h
   LEA DX, msg1
   INT 21h

   ; Read the first operand
   mov ah, 01h
   int 21h
   sub al, '0'
   mov operand1, al
```

```
; Input operator
MOV AH, 09h
LEA DX, msg3
INT 21h

; Read the operator
mov ah, 01h
int 21h
mov operator, al

; Input second operand
MOV AH, 09h
LEA DX, msg2
INT 21h

; Read the second operand
mov ah, 01h
int 21h
sub al, '0'  ; Convert ASCII to binary
mov operand2, al



;Perform the operation based on the operator
cmp operator, '+'
je addition
```

```
    cmp operator, '-'
    je subtraction
    cmp operator, '*'
    je multiplication
    cmp operator, '/'
    je division
    jmp invalid_operator

addition:
    mov al, operand1
    add al, operand2
    mov result, al
    jmp print_result

subtraction:
    mov al, operand1
    sub al, operand2
    mov result, al
    jmp print_result

multiplication:
    mov al, operand1
    mul operand2
    mov result, al
    jmp print_result

division:
    cmp operand2, 0
```

```
    je divide_by_zero
    mov al, operand1
    mov bl, operand2
    mov ah, 0 ; Clear AH for DIV operation

    div bl
    mov result, al
    jmp print_result

divide_by_zero:

    mov ax, 0
    mov es, ax

    mov al, 75h

    mov bl, 4h
    mul bl
    mov bx, ax

    mov si, offset [infinity_msg]
    mov es:[bx], si
    add bx, 2

    mov ax, cs
    mov es:[bx], ax
```

```
    int 75h

    jmp quit_program



print_result:

     ; Display result
    MOV AH, 09h
    LEA DX, msg5
    INT 21h

    ; Check if result is negative
    cmp result, 0
    jns print_result_positive
    ; If negative, print negative sign
    mov dl, negative_sign
    mov ah, 02h
    int 21h
    ; Convert result to positive for ASCII conversion
    neg result

print_result_positive:
    MOV AL, result    ; Load the two-digit number into AL
    MOV AH, 0         ; Clear AH register
    MOV BH, 0         ; Clear BH register to store the tens digit
    MOV BL, 10        ; Load BL with 10 to divide AX by 10
```

```
    DIV BL          ; Divide AX by BL, quotient in AL (tens digit), remainder in AH (ones digit)
    ADD AL, '0'     ; Convert tens digit to ASCII
    MOV ascii[0], AL ; Store tens digit in ASCII representation
    ADD AH, '0'     ; Convert ones digit to ASCII
    MOV ascii[1], AH ; Store ones digit in ASCII representation

    ; Terminate the message string
    MOV BYTE PTR [ascii+2], 0Dh ; Carriage return
    MOV BYTE PTR [ascii+3], 0Ah ; Line feed
    MOV BYTE PTR [ascii+4], '$' ; End of string ('$')

    ; Display ASCII representation
    MOV AH, 09h     ; Function to display a string
    LEA DX, ascii   ; Load the address of the ASCII string
    INT 21h         ; Call DOS interrupt
    jmp quit_program

invalid_operator:

    mov ah, 09h
    lea dx, msg_invalid_operator
    int 21h

quit_program:
    mov ah, 4Ch
    int 21h
```

```
main endp


infinity_msg PROC


    mov ah, 09h
    lea dx, msg_divide_by_zero
    int 21h
    IRET

infinity_msg ENDP

end main
```

Result Analysis:

After evaluating and then running the assembly code with emu8086, it displays the following results that ask for the operation to be performed.

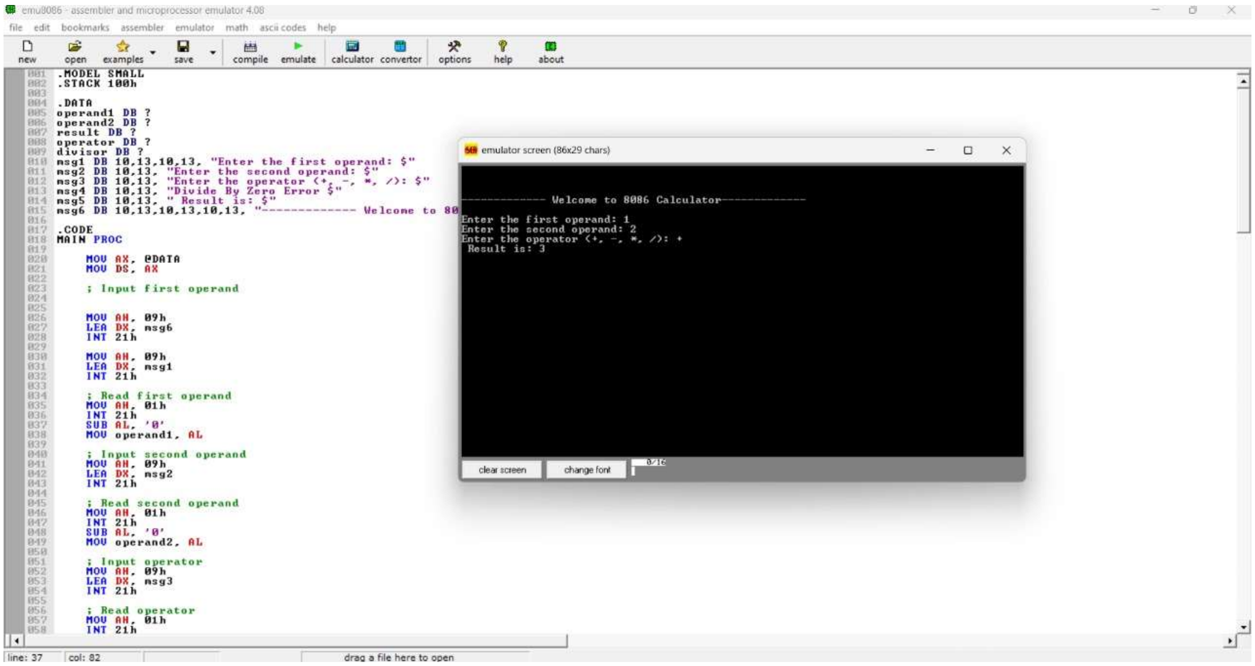

Fig: 6.1: Output display of calculator

- <u>Parts of the Code that Display a Message on Console</u>

The code uses the interrupt INT 21h with function 09h to display messages on the console. Here is how it is done:

MOV AH, 09h ; Function to display a string

LEA DX, msg0 ; Load the address of the message into DX

INT 21h   ; Interrupt to display the string

<u>Explanation</u>

MOV AH, 09h: This sets the function number to 09h, which tells DOS to display a string.

LEA DX, msg0: This loads the effective address of the string msg0 into the DX register. The string should end with a $ character, which is used by DOS to denote the end of the string.

INT 21h: This is the DOS interrupt call that executes the display string function.

The code contains several similar sections for displaying different messages:

```
MOV AH, 09h
LEA DX, msg0
INT 21h

MOV AH, 09h
LEA DX, msg1
INT 21h

MOV AH, 09h
LEA DX, msg3
INT 21h

MOV AH, 09h
LEA DX, msg2
INT 21h

MOV AH, 09h
LEA DX, msg5
INT 21h

MOV AH, 09h
LEA DX, msg_divide_by_zero
INT 21h

MOV AH, 09h
LEA DX, msg_invalid_operator
INT 21h
```

- Parts of the Code that Get Input from Console

The code uses the interrupt INT 21h with function 01h to get input from the console. Here is how it is done:

```
MOV AH, 01h  ; Function to read a character

INT 21h      ; Interrupt to read the character

SUB AL, '0'  ; Convert ASCII to binary

MOV operand1, AL ; Store the result in operand1
```

Explanation

MOV AH, 01h: This sets the function number to 01h, which tells DOS to read a character from the standard input (usually the keyboard).

INT 21h: This is the interrupt call that reads the character. The character read is stored in the AL register.

SUB AL, '0': This converts the ASCII character to its corresponding binary value by subtracting the ASCII value of '0'.

MOV operand1, AL: This stores the binary value in the operand1 variable.

The code contains similar sections for reading different inputs:

```
; Read the first operand

MOV AH, 01h

INT 21h

SUB AL, '0'

MOV operand1, AL


; Read the operator

MOV AH, 01h

INT 21h

MOV operator, AL


; Read the second operand

MOV AH, 01h
```

INT 21h

SUB AL, '0'

MOV operand2, AL

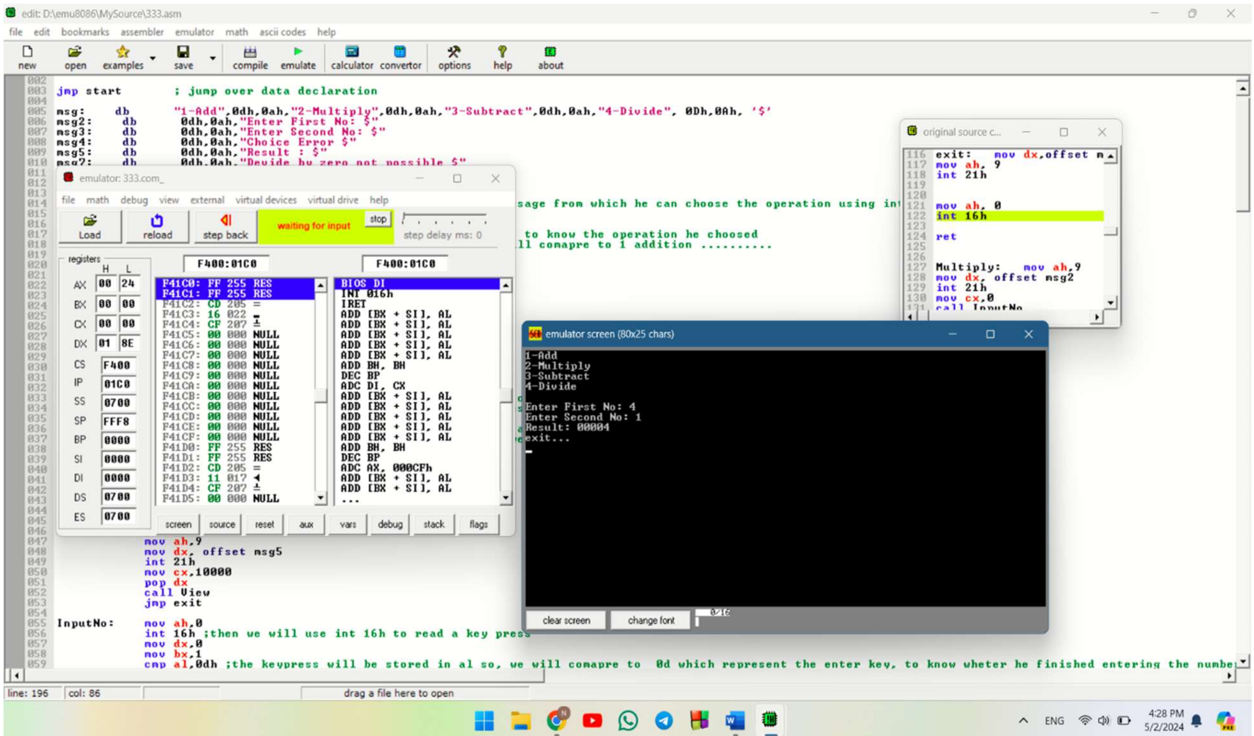

Fig: 6.2: Divide operation & result

When divisor is taken zero, then the program calls for the software interrupt type 33 to display a message that Divide by zero is not possible instead of showing Divide Error (type 0)

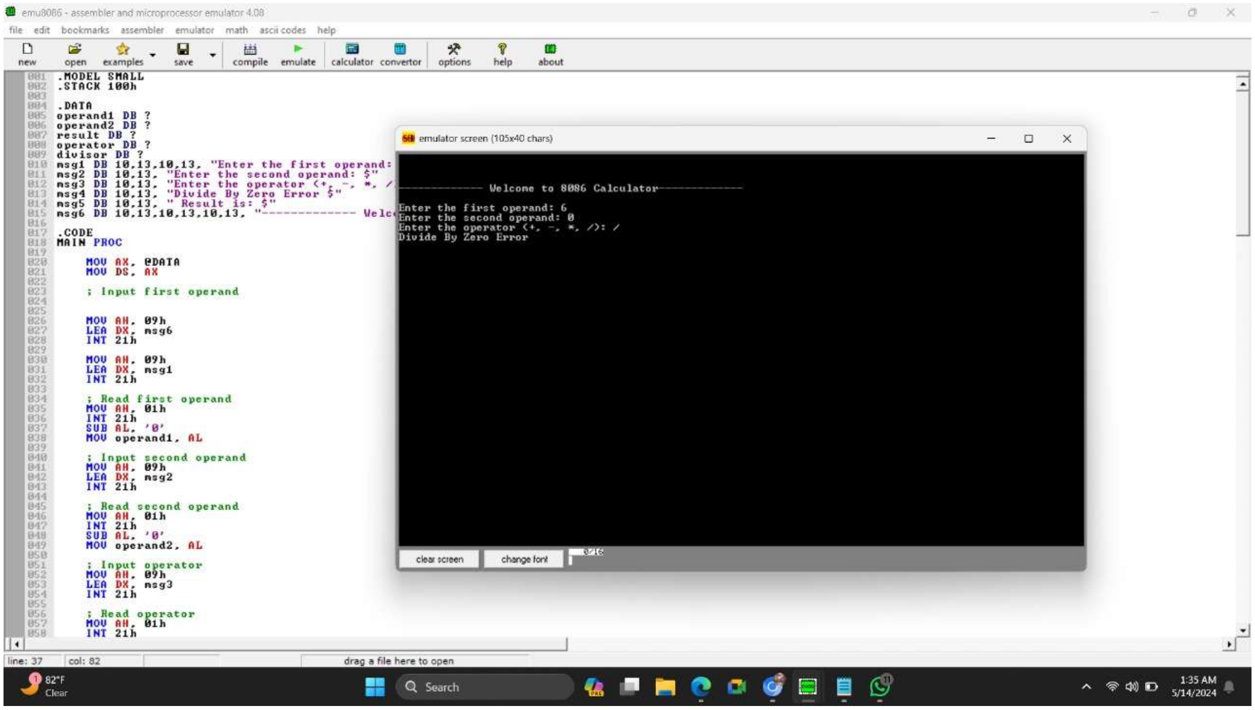

Fig 6.3: Showing Divide by zero error

- <u>Updating the Interrupt Vector Table</u>

The 8086 assembly code for the calculator cannot update the Interrupt Vector Table (IVT) because it lacks the necessary instructions and procedures to access and modify this critical area of memory. The IVT resides in the first 1 KB of memory (addresses 0000h to 03FFh), requiring manipulation of the segment registers to point to this region. Specifically, updating the IVT involves setting the segment register, typically `ES`, to 0000h and writing new interrupt handler addresses directly to specific memory locations. The current code does not include any such operations; it focuses solely on arithmetic computations and input/output handling through DOS interrupts (INT 21h). Without the necessary low-level system programming to alter segment registers and access the IVT, the code cannot modify interrupt vectors or implement custom interrupt service routines.

Discussion:

The 8086-calculator implemented in assembly language demonstrates fundamental arithmetic operations including addition, subtraction, multiplication, and division, efficiently utilizing the capabilities of the 8086 microprocessors. This simple yet functional program reads user input, processes the arithmetic operations, and displays results using basic interrupt calls. While the program achieves its primary objective, it is limited by its current scope, handling only single-digit operands and lacking extensive error handling. One significant limitation is the simplistic approach to division, where division by zero is managed but other invalid inputs are not addressed. This could be enhanced by implementing a more comprehensive error-checking mechanism. Moreover, the interface and interaction could be improved to support multi-digit arithmetic, which would involve modifying the input reading process to handle strings of digits and converting them appropriately. Additionally, the program could be extended to include floating-point arithmetic using the 8087 co-processor, providing a broader range of calculations. Optimization of the code for better readability and maintainability through comments and subroutines would further enhance the program. Despite these limitations, the calculator serves as a robust example of utilizing assembly language for basic arithmetic operations, offering a practical demonstration of the 8086 microprocessor's capabilities and paving the way for further refinements and more sophisticated computational functionalities.

Conclusion:

The main objective of designing an 8086-microprocessor calculator has been successfully achieved, demonstrating fundamental arithmetic operations through efficient use of assembly language. This project highlights the 8086’s ability to handle basic computations, user input, and result display. While the calculator is effective for single-digit operations, future improvements could expand its capabilities to multi-digit arithmetic and enhanced error handling, showcasing the microprocessor's versatility and potential for more complex applications.