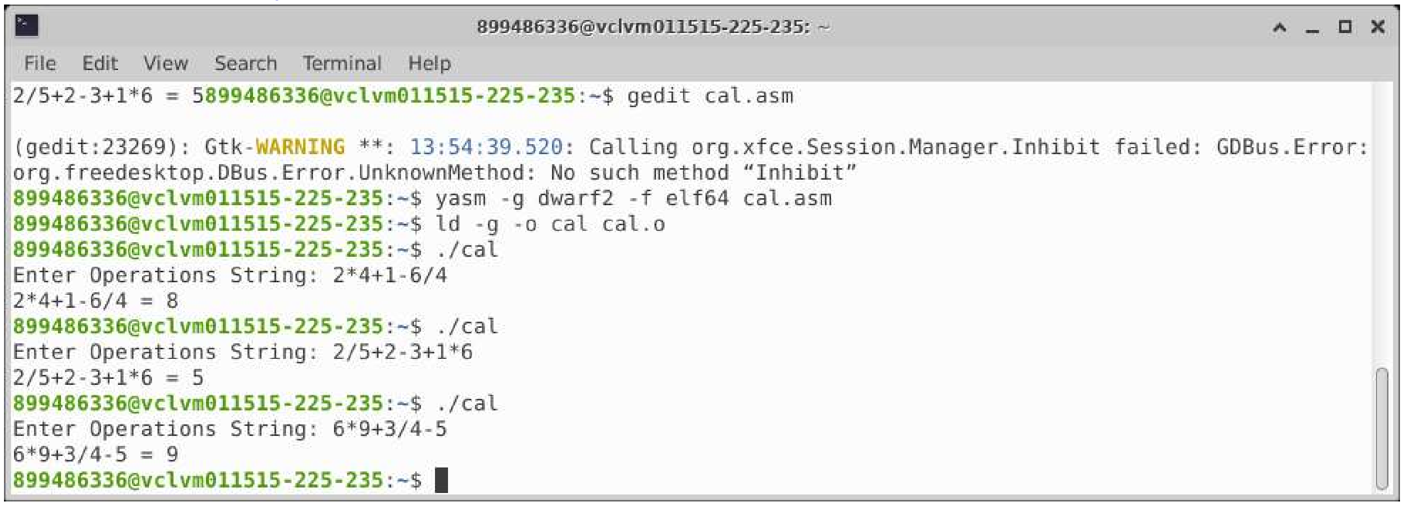
CPSC 240: Computer Organization and Assembly Language
Final Project, Spring Semester 2024

CWID: 885097022 Name: Zeid Aldaas

- Use YASM/NASM in Linux system for assembly language programming.
- Design a one-digit integer calculator. The program reads a constant equation from keyboard input, calculates the result, and display the equation and result to the Terminal Window.
- Operations shall include at least one addition, one subtraction, one multiplication, and one division, but they can be entered in a different order. For example, a+b*c/d-e or a*b-c+d/e, where a, b, c, d, and e are integers from 0 to 9 entered on the keyboard. The more operations a program provides, the higher the program's score. Note: [1]This operation is performed from left to right, without the need for multiplication and division followed by addition and subtraction. [2]Whenever possible, enter equations that evaluate to positive integers, since the display of negative integers was not discussed this term.
- Prepare a detailed report of one-digit integer calculator according to the report document and save the report in pdf format. The more detailed the documentation, the higher the documentation score.
- The grading of the final project will be broken even by two parts, 50% of the program and 50% of the report.
- Submit the document (.pdf file) and source code (.asm file) to Canvas before the deadline.
- Due: 23:59PM, Sunday (May 12, 2024), in Canvas only.
- Late final project will not be accepted.

Simulation Samples:


899486336@vclvm011515-225-235: ~
File Edit View Search Terminal Help

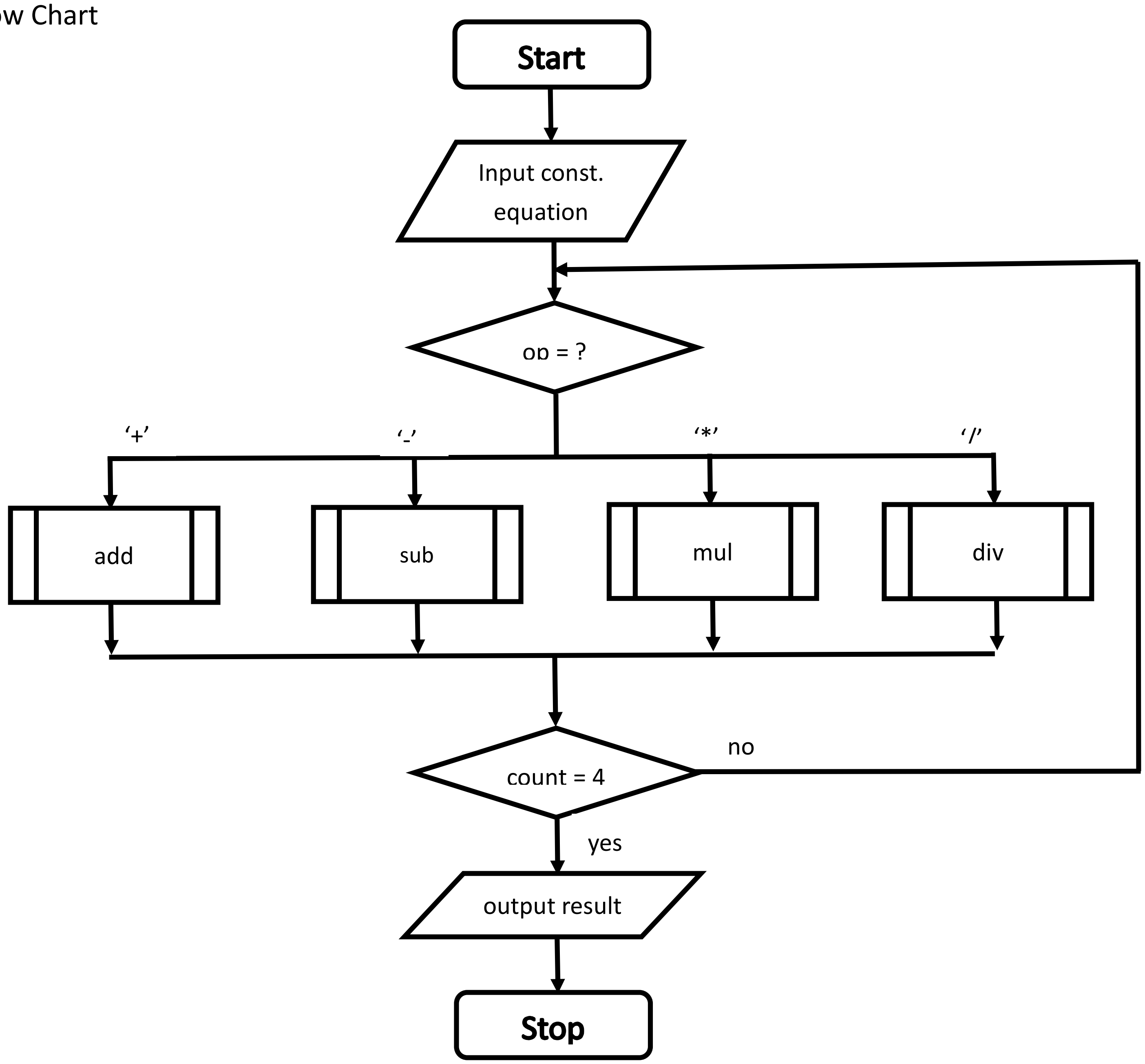
```
2/5+2-3+1*6 = 5899486336@vclvm011515-225-235:~$ gedit cal.asm

(gedit:23269): Gtk-WARNING **: 13:54:39.520: Calling org.xfce.Session.Manager.Inhibit failed: GDBus.Error:
org.freedesktop.DBus.Error.UnknownMethod: No such method "Inhibit"
899486336@vclvm011515-225-235:~$ yasm -g dwarf2 -f elf64 cal.asm
899486336@vclvm011515-225-235:~$ ld -g -o cal cal.o
899486336@vclvm011515-225-235:~$ ./cal
Enter Operations String: 2*4+1-6/4
2*4+1-6/4 = 8
899486336@vclvm011515-225-235:~$ ./cal
Enter Operations String: 2/5+2-3+1*6
2/5+2-3+1*6 = 5
899486336@vclvm011515-225-235:~$ ./cal
Enter Operations String: 6*9+3/4-5
6*9+3/4-5 = 9
899486336@vclvm011515-225-235:~$
```

Report Documentation:

I. Cover page
    a) School name
    b) Department name
    c) Project name
    d) Student name and CWID
    e) Document date
II. Introduction
    a) Introduce the difference between keyboard input (ASCII code) and decimal number.
    b) Introduce the difference between decimal number and monitor output (ASCII code).
    c) Introduce the addition, subtraction, multiplication, and division in the assembly language.
III. Design Principle (Algorithm)
    a) Provide a flow chart and entire assembly program with comments.
    b) Provide the conversion principle (algorithm) of keyboard input to decimal numbers and symbols.
    c) Provide the conversion principle (algorithm) of decimal number to ASCII code.
IV. Simulation Results
    a) Run at least three simulations and take snapshots of the Terminal Emulator. Append these pictures to the simulation results of the document.
    b) Verify the simulation results and interpret the results.
V. Conclusion
    a) Write a conclusion.

Final project grading is distinct from assignments, quizzes, and final exams. For quizzes and exams, you will receive full marks if students answer correctly. For homework, even if the student's program has a big mistake, I will only deduct a little point. However, the grading of the final project should be compared with other students in our class. The best report gets 100 points, while other students' reports get only a percentage of the best report based on the comparison with the best report. However, it is usually higher than or equal to 70 points. If the submitted report differs too much from the requirements of the final project, the report may get below 70 points.

Flow Chart


Start
Input const. equation
op = ?
'+'
'-'
'*'
'/'
add
sub
mul
div
count = 4
no
yes
output result
Stop

Cover Page

California State University, Fullerton
Computer Science
Final Assembly Project – Calculator
Zeid Aldaas – 885097022
5/12/2024

Introduction

a) In the context of a calculator program, keyboard input and decimal numbers represent different data types and encodings. When a user types a number on the keyboard, the keyboard sends a signal to the computer, which is then interpreted as an ASCII (American Standard Code for Information Interchange) code. ASCII is a character encoding standard that represents text in computers and other devices. Each character, including digits, is assigned a unique numerical value. For example, the ASCII code for the character '0' is 48, '1' is 49, and so on up to '9', which is 57.

   Decimal numbers, on the other hand, are numerical values represented in the base-10 numbering system. They are used in arithmetic operations and are not confined to character encoding. When a digit is entered via the keyboard, the program needs to convert the ASCII code to its corresponding decimal value for proper arithmetic computation. For instance, converting the ASCII code 48 ('0') to the decimal value 0 involves subtracting 48 from the ASCII code.

b) After performing calculations, the results need to be converted back to a format that can be displayed on the monitor. The monitor output still relies on ASCII codes. Thus, converting a decimal number back to an ASCII code involves adding 48 to the decimal value. For instance, converting the decimal number 5 to its ASCII representation involves adding 48, yielding ASCII code 53, which represents the character '5'.

   To display a number like 25 on the monitor, the program needs to handle each digit separately. First, it divides the number by 10 to isolate the digits. For 25, it first finds the quotient 2 and remainder 5. It then converts each digit to its ASCII equivalent (2 to 50 and 5 to 53) and outputs these characters sequentially.

c) Arithmetic operations in assembly language involve direct manipulation of CPU registers. Here’s a brief overview of how addition, subtraction, multiplication, and division are implemented in assembly language:

Addition

Addition in assembly language involves adding the contents of two registers or a register and an immediate value. The result is usually stored in one of the registers involved in the operation. For example:

```
mov eax, 5        ; Load the value 5 into register eax
add eax, 3        ; Add the value 3 to the contents of eax, result in eax (8)
```

Subtraction

Subtraction works similarly to addition but subtracts the contents of one register or an immediate value from another register. The result is stored in one of the registers involved:

```
mov eax, 5        ; Load the value 5 into register eax
sub eax, 3        ; Subtract 3 from the contents of eax, result in eax (2)
```

Multiplication

Multiplication in assembly language typically involves the mul or imul instructions. The mul instruction is used for unsigned multiplication, while imul is for signed multiplication. The result of the multiplication is stored in the eax register:

```
mov eax, 5        ; Load the value 5 into register eax
mov ebx, 3        ; Load the value 3 into register ebx
imul ebx          ; Multiply the contents of eax by the contents of ebx, result in eax (15)
```

Division

Division in assembly language is performed using the div or idiv instructions. div is used for unsigned division, and idiv for signed division. The dividend is placed in the eax register, and the divisor in another register. The quotient is stored in eax, and the remainder in edx:

```
mov eax, 10       ; Load the value 10 into register eax (dividend)
mov ebx, 2        ; Load the value 2 into register ebx (divisor)
xor edx, edx      ; Clear edx to ensure no residual value affects division
div ebx           ; Divide eax by ebx, quotient in eax (5), remainder in edx (0)
```

Design Principle (Algorithm)

a) Below is both a written and visual flowchart for calculator.asm. Calculator.asm is attached to the document submission.

<u>Written Flowchart</u>

1. **Start -** Initialize necessary registers and memory locations.

2. **Display Prompt -** Print the message "Enter Operations String: " using the '**sys_write**' system call:

```
mov eax, 4                  ; sys_write
mov ebx, 1                  ; file descriptor (stdout)
mov ecx, prompt
mov edx, 25                 ; length of prompt string
int   0x80
```

3. **Read Input -** Read the user's input string using the '**sys_read**' system call:

```
mov eax, 3                  ; sys_read
mov ebx, 0                  ; file descriptor (stdin)
mov ecx, input
mov edx, 256                ; buffer length
int 0x80
```

4. **Initialize Result -** Set the initial result to the first number in the input by calling '**next_number**':

```
mov esi, input
call  next_number           ; Get first number as initial result
mov [result], eax
```

5. **Parse Loop -** Loop through each character in the input.

   - **Check For Newline or Null Terminator**:

   ```
   lodsb
   cmp al, 10                  ; Newline
   je    print_result
   cmp al, 0                   ; Null terminator
   je    print_result
   ```

   - **Identify Operator and Evaluate Operation by calling 'evaluate_operation'**:

     - **Addition**:

     ```
     cmp     al, '+'
     je add_operation
     ```

     - **Subtraction**:

```
cmp     al, '-'
je sub_operation
```

- **Multiplication**:

```
cmp     al, '*'
je mul_operation
```

- **Division**:

```
cmp     al, '/'
je div_operation
```

- **Invalid Character**:

```
jmp  error
```

6. **Perform Operation** – Convert number and perform operation in accordance to which operator was identified.

- **Convert the ASCII Character Into its Corresponding Decimal Value by Calling 'next_number'**:

```
lodsb
sub  al, '0'
cmp al, 0
jl      error
cmp al, 9
jg     error
movzx      eax, al
ret
```

- **Perform Operation**:
  - **Addition (Also Check for Overflow Error)**:

```
call  next_number
add  [result], eax
jo     overflow_output
ret
```

  - **Subtraction (Also Check for Overflow Error)**:

```
call  next_number
sub  [result], eax
jo     overflow_output
ret
```

  - **Multiplication (Also Check for Overflow Error)**:

```
call  next_number
mov ebx, [result]               ; Load the value from result into ebx
imul ebx, eax                   ; Multiply ebx by eax
jo    overflow_output
mov [result], ebx               ; Store the result back into the memory location
ret
```

- **Division (Also Check for Divide by Zero & Overflow Error)**:

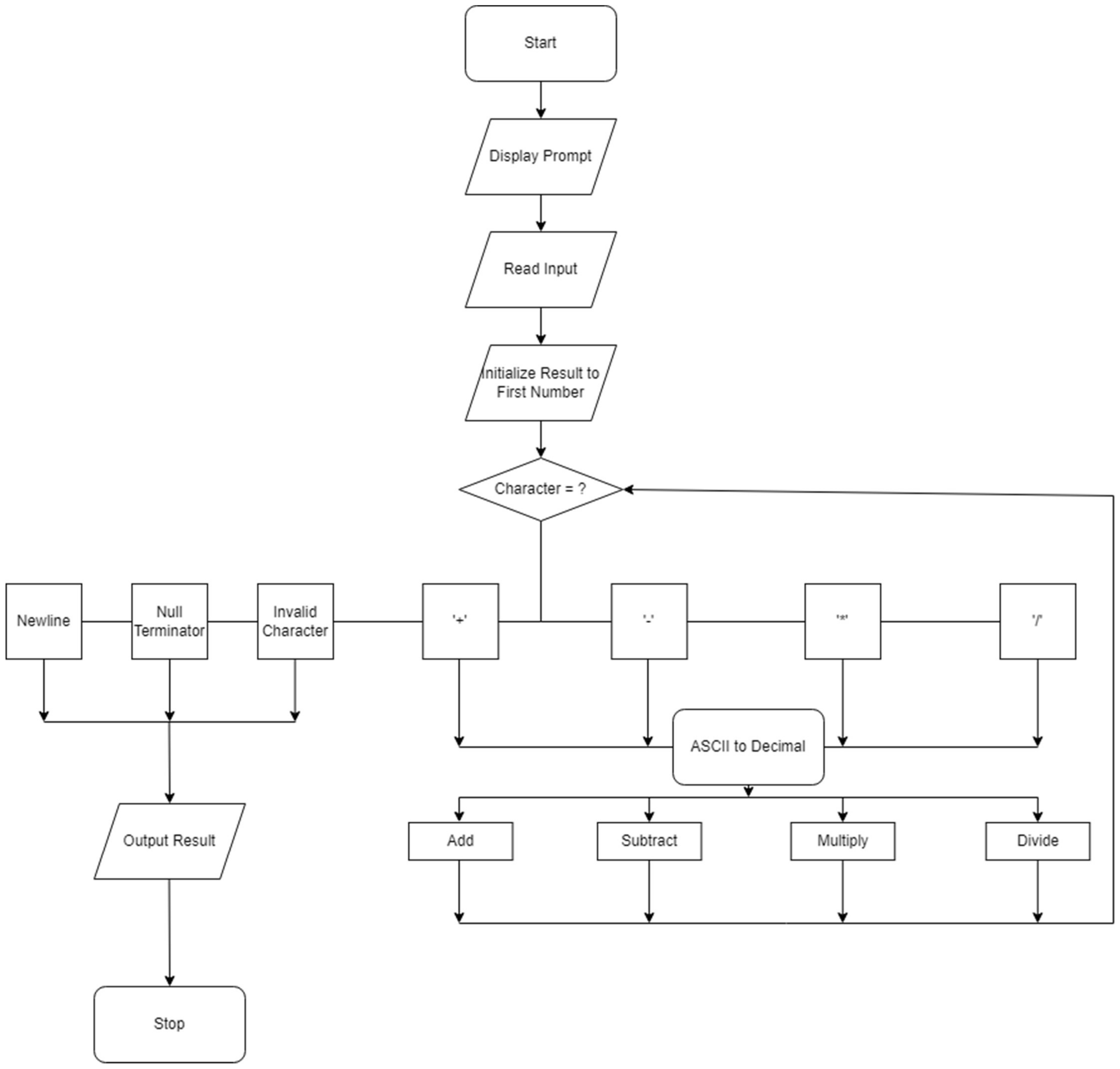
```
call  next_number
mov ebx, eax                    ; Store the next number in ebx (divisor)
test  ebx, ebx                  ; Check if the divisor is zero
jz    div_zero_output           ; Jump if zero

mov eax, [result]               ; Load the current result into eax (dividend)
xor   edx, edx                  ; Clear edx for division
div   ebx                       ; Divide eax by ebx
jo    overflow_output
mov [result], eax               ; Store the result back
ret
```

7. **Print Result** - Print equation and result (converted back to ASCII) or print error message.

8. **Exit Program**:

```
mov         rax, SYS_exit               ;terminate excuting process
mov         rdi, EXIT_SUCCESS           ;exit status
syscall                                 ;calling system services
```

Visual Flowchart



b) The conversion principle describes how the program interprets keyboard input into decimal numbers and symbols that the assembly language can operate on. This involves two main processes: extracting numbers and detecting operations. The conversion uses ASCII values to determine the input character's nature—whether it's a number or an operator.

1. **Numeric Input Conversion**

- Each character from the input is read sequentially.

- The ASCII value of '0' (which is 48) is subtracted from the ASCII value of the character.
- If the resulting value is between 0 and 9 (inclusive), it is considered a valid single-digit integer.
- This integer is then used directly in calculations.

2. **Operation Symbol Detection**
   - Each character is compared against ASCII values of known operation symbols ('+', '-', '*', '/').
   - If a match is found, the corresponding operation is queued for execution in the order they appear in the input, following the straight-line sequence of execution.

In calculator.asm:

1. **Numeric Input Conversion**
   a. **Read Character**: Read a character from the input string using lodsb.
   b. **Subtract ASCII Offset**: Subtract 48 (the ASCII value of '0') from the character's ASCII code.
   c. **Store Decimal Value**: Store the result as the decimal value of the character in the eax register.

   This is done in '**next_number**':

```
next_number:
    ; Extract the next numeric value (single digit 0-9)
    lodsb
    sub  al, '0'
    cmp  al, 0
    jl   error
    cmp  al, 9
    jg   error
    movzx    eax, al
    ret
```

2. **Operation Symbol Detection**
   a. **Read Character**: Read a character from the input string using lodsb.
   b. **Compare ASCII Values**: Compare it against ASCII values of known operators ('+', '-', '*', '/').
   c. **Perform Operation**: Based on the identified operator, perform the corresponding arithmetic operation using the next number in the input string.

This is done in '**evaluate_operation**':

```
evaluate_operation:
        cmp al, '+'
        je     add_operation
        cmp al, '-'
        je     sub_operation
        cmp al, '*'
        je     mul_operation
        cmp al, '/'
        je     div_operation
        jmp  error
```

c) Converting decimal numbers to ASCII code is a fundamental operation in assembly programming, especially when dealing with numerical outputs that need to be displayed as part of a user interface. Each digit of a decimal number is represented in the computer by its ASCII equivalent. The ASCII values of the decimal digits 0 through 9 range from 48 to 57. To convert a decimal digit to its ASCII representation, you simply add 48 to the digit.

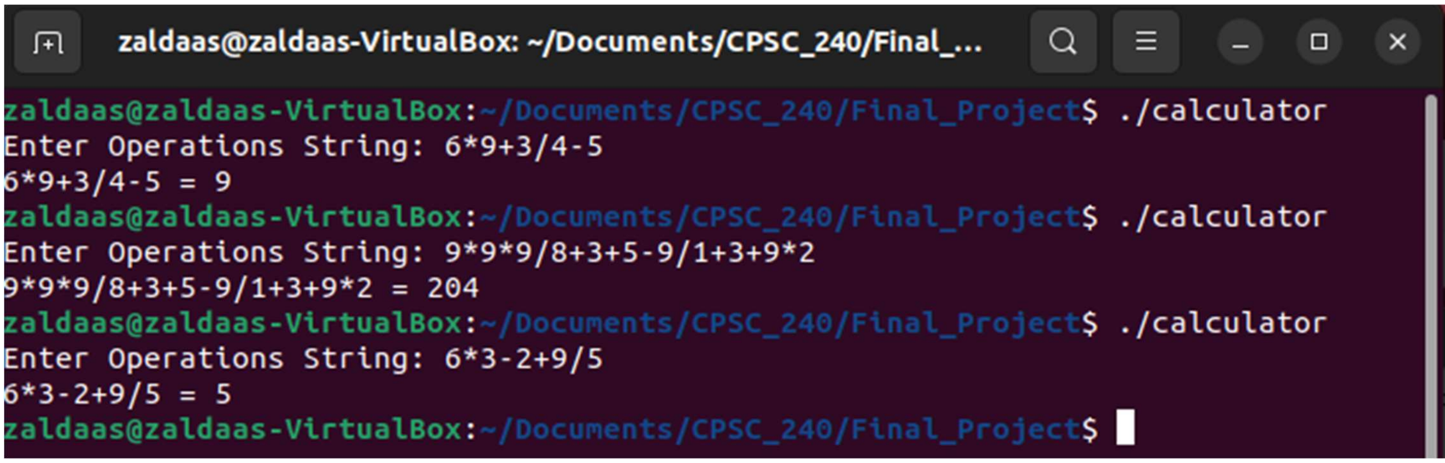
In calculator.asm:

1. **Isolate Digits**: Divide the number by 10 to isolate each digit.
2. **Add ASCII Offset**: Add 48 to each digit to convert it to its ASCII code.
3. **Store/Display ASCII Characters**: Store or display the resulting ASCII characters.

```
print_integer:
    ; Convert integer in EAX to string representation
    mov edi, 10
    mov ecx, newline + 1                ; Start writing at newline position + 1
    mov byte [ecx], 0

print_integer_loop:
    xor   edx, edx
    div   edi
    add  dl, '0'
    dec   ecx
    mov [ecx], dl
    test  eax, eax
    jnz   print_integer_loop

    ; Append newline after the integer string
    mov byte [newline + 1], 10          ; newline character
    mov byte [newline + 2], 0           ; null terminator

    ; Print the integer string with newline
    mov eax, 4
mov  ebx, 1
lea  edx, [newline + 3]        ; point just past the newline char
sub  edx, ecx                  ; calculate length of number + newline
int  0x80
ret
```

Simulation Results

a) **Valid Equation/Result Simulation & Verification**:



```
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 6*9+3/4-5
6*9+3/4-5 = 9
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 9*9*9/8+3+5-9/1+3+9*2
9*9*9/8+3+5-9/1+3+9*2 = 204
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 6*3-2+9/5
6*3-2+9/5 = 5
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$
```

Equation 1: 6 * 9 + 3 / 4 - 5

1. 6 * 9 = 54
2. 54 + 3 = 57
3. 57 / 4 = 14.25 (rounded down to 14)
4. 14 - 5 = 9

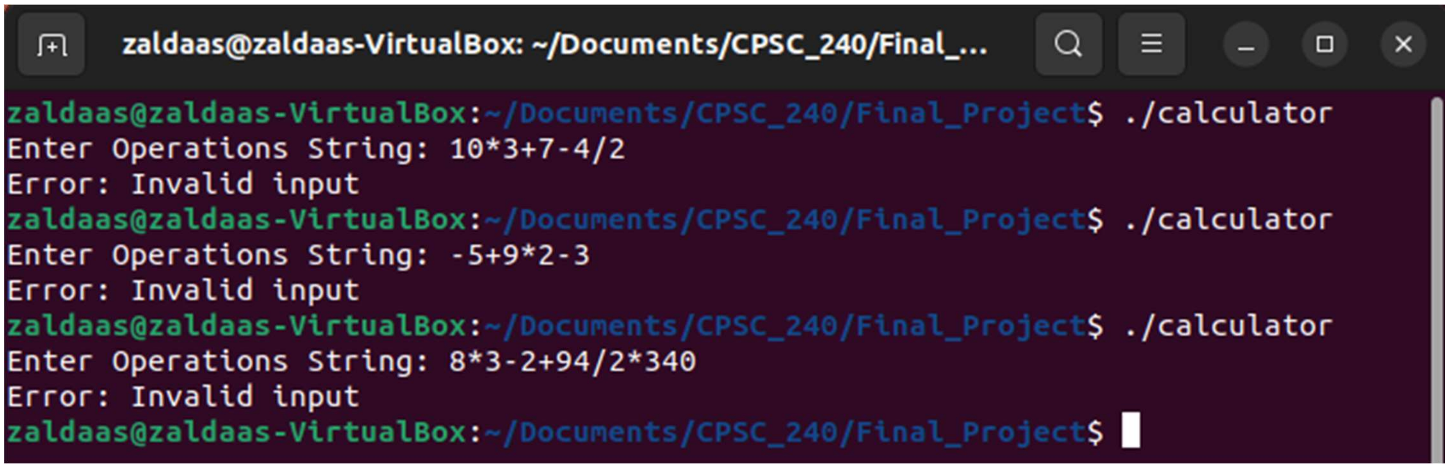
Equation 2: 9 * 9 * 9 / 8 + 3 + 5 - 9 / 1 + 3 + 9 * 2

1. 9 * 9 = 81
2. 81 * 9 = 729
3. 729 / 8 = 91.125 (rounded down to 91)
4. 91 + 3 = 94
5. 94 + 5 = 99
6. 99 - 9 = 90
7. 90 / 1 = 90
8. 90 + 3 = 93
9. 93 + 9 = 102
10. 102 * 2 = 204

Equation 3: 6 * 3 - 2 + 9 / 5

1. 6 * 3 = 18
2. 18 - 2 = 16
3. 16 + 9 = 25
4. 25 / 5 = 5

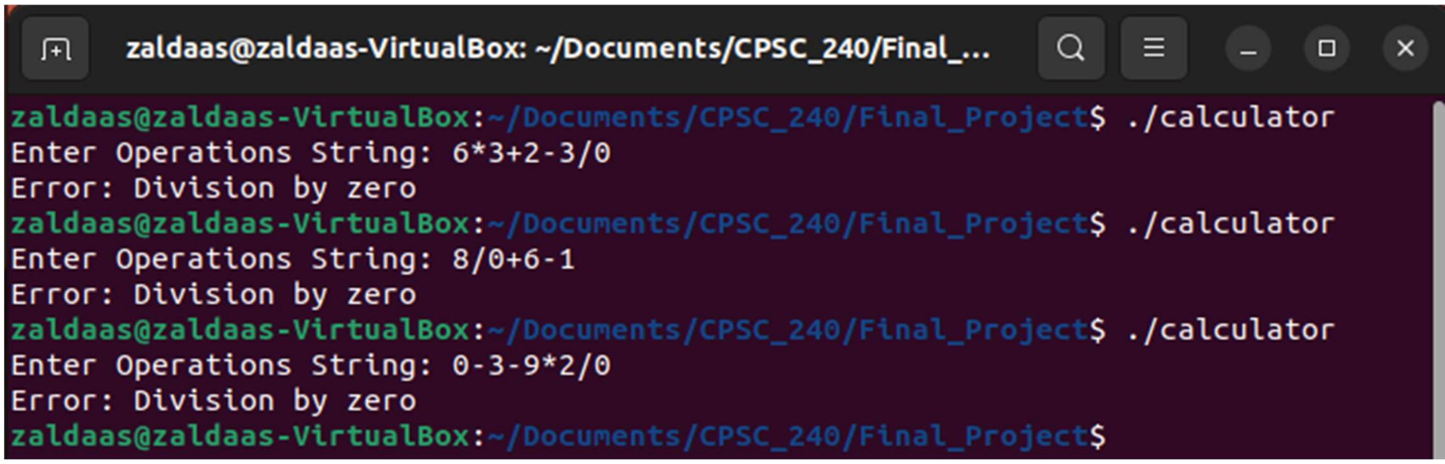
b) **Invalid Input Error (Multi-Digit Number/Negative Number) Simulation & Verification**:



```
zaldaas@zaldaas-VirtualBox: ~/Documents/CPSC_240/Final_...
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 10*3+7-4/2
Error: Invalid input
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: -5+9*2-3
Error: Invalid input
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 8*3-2+94/2*340
Error: Invalid input
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$
```

As shown, the error check works whether the multi-digit number is at the beginning or anywhere else in the equation. In equation 1, the 10 causes the error, in equation 2, -5 causes the error, and in equation 3, 94 and 340 will cause the error.

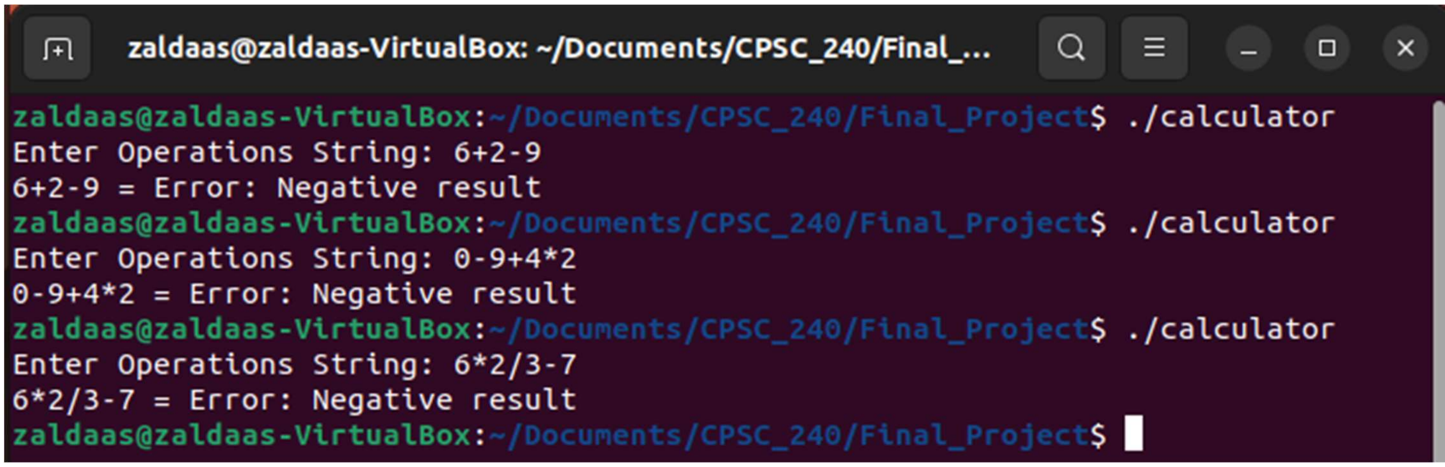
c) **Divide By Zero Error Simulation & Verification**:



```
zaldaas@zaldaas-VirtualBox: ~/Documents/CPSC_240/Final_...
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 6*3+2-3/0
Error: Division by zero
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 8/0+6-1
Error: Division by zero
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 0-3-9*2/0
Error: Division by zero
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$
```

As seen here, the error check considers all cases and is robust. Whether the division by zero occurs at the end of the equation such as in equation 1, or at the beginning of the equation such as in equation 2, it will cause the error. Furthermore, even if it divides a **negative result** by zero, it will return this error as seen in equation 3, rather than the next error check we will discuss.

d) **Negative Answer Error Simulation & Verification**:



```
zaldaas@zaldaas-VirtualBox: ~/Documents/CPSC_240/Final_...
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 6+2-9
6+2-9 = Error: Negative result
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 0-9+4*2
0-9+4*2 = Error: Negative result
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 6*2/3-7
6*2/3-7 = Error: Negative result
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ 
```

Equation 1:

1. 6 + 2 = 8
2. 8 - 9 = -1

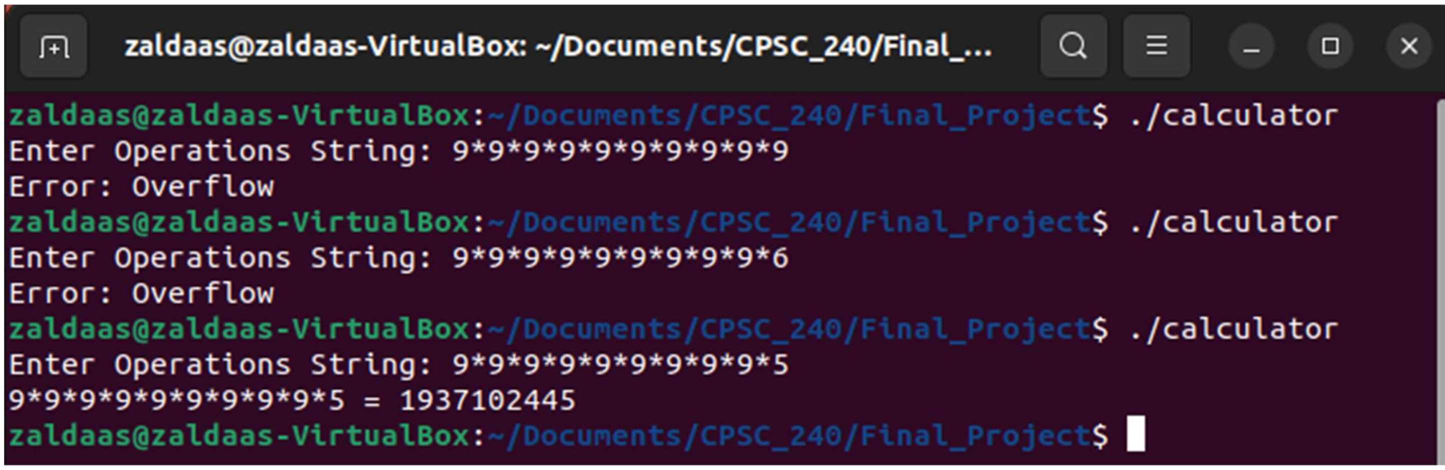
Equation 2:

1. 0 - 9 = -9
2. -9 + 4 = -5
3. -5 * 2 = -10

Equation 3:

1. 6 * 2 = 12
2. 12 / 3 = 4
3. 4 - 7 = -3

e) **Overflow Error Simulation & Verification**:



```
zaldaas@zaldaas-VirtualBox: ~/Documents/CPSC_240/Final_...
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 9*9*9*9*9*9*9*9*9*9
Error: Overflow
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 9*9*9*9*9*9*9*9*9*6
Error: Overflow
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ ./calculator
Enter Operations String: 9*9*9*9*9*9*9*9*9*5
9*9*9*9*9*9*9*9*9*5 = 1937102445
zaldaas@zaldaas-VirtualBox:~/Documents/CPSC_240/Final_Project$ 
```

The maximum positive value for a 32-bit integer is 2,147,483,647, thus meaning any value over this would result in an Overflow error.

Equation 1:

1. 9 * 9 = 81
2. 81 * 9 = 729
3. 729 * 9 = 6561
4. 6561 * 9 = 59049
5. 59049 * 9 = 531441
6. 531441 * 9 = 4782969
7. 4782969 * 9 = 43046721
8. 43046721 * 9 = 387420489
9. 387420489 * 9 = 3486784401

Result: Error - Overflow

Equation 2:

1. 9 * 9 = 81
2. 81 * 9 = 729
3. 729 * 9 = 6561
4. 6561 * 9 = 59049
5. 59049 * 9 = 531441
6. 531441 * 9 = 4782969
7. 4782969 * 9 = 43046721
8. 43046721 * 9 = 387420489
9. 387420489 * 6 = 2324522934

Result: Error - Overflow

Equation 3:

1. 9 * 9 = 81

2. 81 * 9 = 729
3. 729 * 9 = 6561
4. 6561 * 9 = 59049
5. 59049 * 9 = 531441
6. 531441 * 9 = 4782969
7. 4782969 * 9 = 43046721
8. 43046721 * 9 = 387420489
9. 387420489 * 5 = 1937102445

Result: Valid

Conclusion

The assembly language calculator program demonstrates a robust understanding of assembly language operations, including handling user input, performing arithmetic calculations, and converting between ASCII and decimal formats. Each step, from reading the input to performing calculations and displaying results, is meticulously managed using low-level CPU instructions.

The implementation effectively handles errors such as invalid input, division by zero, and arithmetic overflow, ensuring reliable program operation. This project showcases the precision and efficiency achievable with assembly language programming, providing a solid foundation for further exploration of low-level programming concepts and techniques.

The detailed explanations and flowcharts presented offer a comprehensive view of the program's functionality, highlighting the critical steps and corresponding code segments that drive the calculator's operation. This level of detail ensures a thorough understanding of the program's inner workings, making it a valuable learning tool for those interested in assembly language programming.