Experiment 3

Assembly Program – A Simply Calculator

By Domenic Mancuso

University of Southern Maine

Spring 2024

COS 255 Computer Organization Lab

Contents

Problem

In this experiment, you will learn how to use assembly language to create a simple calculator. Your assembly program should be designed to perform basic arithmetic operations based on user input. The program must support the following functionalities:

1. User Input: The user should be able to input two numbers, representing the operands for the arithmetic operation.
2. Addition: Your program should be capable of executing the addition operation on the two input numbers.
3. Subtraction: Implement functionality to subtract the second operand from the first, returning the difference.
4. User Choice: The program should prompt the user to choose the desired operation. The program will execute the corresponding operation based on the user's input: typing '1' triggers addition, '2' for subtraction, and '3' to exit the program directly.
5. Loop Execution: The program will continuously loop, allowing the user to perform multiple operations without restarting the program. This loop will persist until the user types in '3' to exit the program.

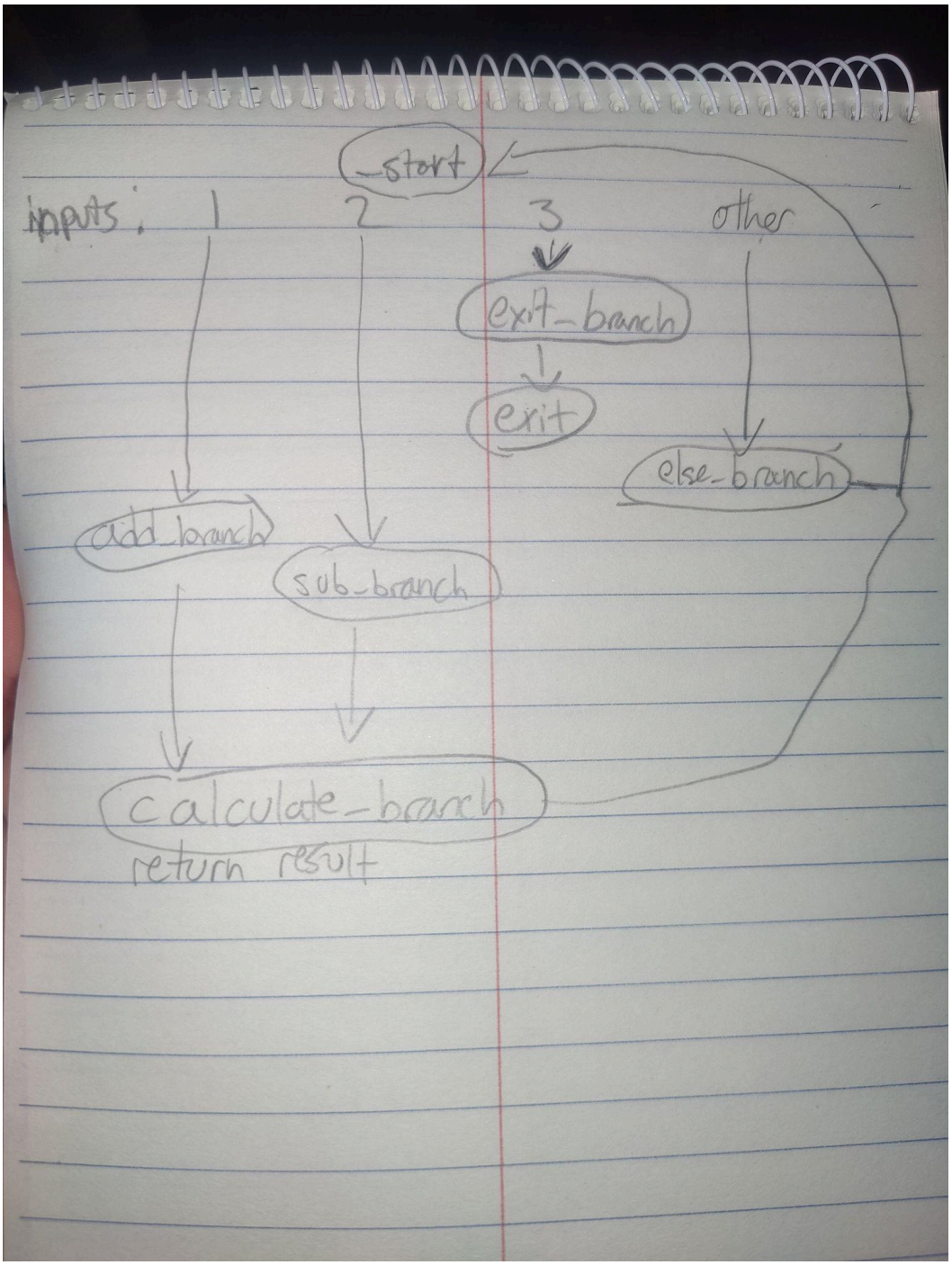
Implementation

Providing a detailed description of your implementation plan that includes the specifics of the if-else conditions, the structure of loops, the criteria for exit conditions, and more. It is highly recommended to supplement your explanation with a flowchart for a visual representation of the logical sequence and interactions within your plan.

I had a pretty straightforward idea of how I was going to solve this experiment from the get-go. I was aware of the loop structures of NASM programs and did not see much trouble in this implementation. NASM's cmp function was a great help to me, since learning how to take string inputs from the previous assignment 5 made a compare sequence possible. The way that I had this plan structured was to save the input stored in inputBuffer from the ASM file into the eax register and run it through four compare statements: the first one will compare eax to 1, if the condition is met, since the key for add is 1, jump to the add loop. If the condition is not met, go down a line where eax is compared to 2. If this is met, jump to the subtract loop. If not, jump to the next line, where eax is compared to 3. If this condition is met, it exits the program. If none of these conditions are met, the program jumps to what I called the else_branch, where a log is printed asking the user to print a valid input, since none of the three conditions were met. The program will then jump back to the _start loop, prompting the user all over again. Since it is a loop and I had my jump statements sorted, I knew that it would not be an issue.

During the implementation of my program, I was wondering about how to handle the actual calculation. I figured that creating a calculation loop would be the easiest approach, since either operation could

store its result in the same variable that could be used in calculation_loop. This worked! When the calculation loop is completed, jump to the _start loop and start over again.



Results

Include screenshots in this section. Ensure the screenshot displays a comprehensive set of test cases that adequately verify addition and subtraction operations, the loop, as well as the exit condition. This is essential to demonstrate the functionality and reliability of your solution.

```
PS C:\Users\every\Desktop\cos 250\asm stuff\experiment 3> ./calc1.exe
Type in 1 for add operation, 2 for subtract, otherwise input 3 to exit
1
Enter add operand one:
5
Enter add operand two:
4
Calculating from add sequence...
9
Type in 1 for add operation, 2 for subtract, otherwise input 3 to exit
2
Enter subtract operand one:
9
Enter subtract operand two:
8
Calculating from subtract sequence...
1
Type in 1 for add operation, 2 for subtract, otherwise input 3 to exit
4
Invalid input, please try again
Type in 1 for add operation, 2 for subtract, otherwise input 3 to exit
3
Thanks for playing!
PS C:\Users\every\Desktop\cos 250\asm stuff\experiment 3>
```

This screenshot showcases the functionality of every operation in my program. Following the experiment's conditions, I did not consider if operands or the calculation result is more than 10. This proved to be an issue as well because if a number greater than or equal to 10 or less than 0 was calculated, a symbol gets printed:

```
Enter add operand one:
8
Enter add operand two:
8
Calculating from add sequence...
@
Type in 1 for add operation, 2 for subtract, otherwise input 3 to exit
2
Enter subtract operand one:
8
Enter subtract operand two:
9
Calculating from subtract sequence...
/
```

Another notable bug was that adding digits greater than or equal to 10 would add the first digit only. I assume this is because only 1 byte got stored in the pointer to the message's length, which could have been solved by making its size greater.

Conclusion

Did you encounter any challenges during the execution of this experiment? If yes, state the issue and explain how to solve it. If not, elaborate on the insights gained from this experiment.

I had only three large challenges when writing this program. Since I did not want to overwhelm myself when starting this program, I decided to not put any code into the subtract branch and instead, add a jump statement to _start so that I could focus on my add method.

1) '?1' problem: My first issue consequently happened in the add_branch when I figured out how to add and convert the two inputs into a digit. The result that was provided to calculate_branch would print the correct result (as long as result was >10), but it would always print "?1" after the digit, no matter what inputs I gave, even if it was greater than 10.

Looking back at how I solved this problem, it may have not been a complete solution, but it did the job required considering due parameters (input or output must not be greater than or equal to 10). I will paste my code that was giving me an issue:

```
calculate_branch:
    mov [resultChar], eax


    push -11
    call _GetStdHandle@4
    mov [stdHandle], eax


    push 0
    push charsWrite
    push 1
    push resultChar
    push dword [stdHandle]
    call _WriteConsoleA@20
```

after pushing charsWrite, I had a large number of bytes pointed to the length of the message. Instead of 1, I pushed 128. I figured that was an issue so I changed it to push 1 since we are only printing 1 digit (1 byte) in length. This solved the problem but if the calculation result was greater than 1 byte it will not print the answer that you want.

2) newLine problem: My second issue was happening at the same time as my first issue, I needed to tackle one problem at a time. When the result is printed, when _start loop commences, there would be no new line space between the two outputs, result would be concatenated to the initial message in _start without any space between the two. To solve this problem, I initialized a newLine variable with its message length pointer. Here is my code that showcases the fix, also in calulate_branch:

```
    push 0

    push charsWrite

    push newlineLen

    push newline

    push dword [stdHandle]

    call _WriteConsoleA@20


jmp _start
```

As you can see, it was a simple fix. After newLine is printed, the program jumps back to start and the loop begins again.

3) subtract problem: When working on the add method, it took a bit of fiddling around to convert the two inputs to usable data (i.e., not printing ASCII symbols). Once this was fixed, my code at the end of add branch looked like this:

```
;perform calculation

    mov eax, [input1]

    sub eax, '0'

    add eax,  [input2]

jmp calculate_branch
```

When this was figured out, I pasted the entirety of my add method into the subtract_branch. Next, I changed add eax, [input2] to sub eax, [input2]. This was not helpful to me because when a subtraction was processed, it would print even weirder symbols in the calculate_branch, such as pi!

After a bit more fooling around, I had a feeling that it might be something simpler than I thought; here is my code for the subtract_branch:

```
    mov eax, [input1]

    add eax, '0'

    sub eax,  [input2]

    jmp calculate_branch
```

As you can see, I had to do a sort of reverse operation on eax to convert its values to a digit. I am still at a loss for how it worked but my conclusion is that a double subtract operation on eax will have a “double negative” effect, converting eax’s value to a digit, then back to an ASCII value. I believe this logic sticks because the opposite sequence is committed when converting the add_branches value in eax to a digit.

Overall, these three issues gave me the most problems but they were not terrible. I believe that at this point I have a good understanding of the assembly language as this program took me two days to complete.