California State University, Fullerton
Department of Computer Science and Engineering

Assembly Calculator: A One-Integer Arithmetic Program

Andrew Saldana (886880327)
CPSC 240
14 May 2023

Introduction:

When receiving input from a keyboard, it is received by the computer as ASCII code. ASCII code represents characters as numerical or hexadecimal values. Each character on the standard keyboard contains its own unique ASCII code. For example, entering ‘Q’ on the keyboard will yield the decimal value of 81 to be interpreted by the computer. When it comes to decimal numbers, they are numerical values which represent quantities used in mathematical calculations. For example, the decimal number 44 simply represents the quantity 44.

When user input is received in the form of a numerical expression, that input is received as a series of character values. Each character in the expression is represented by its corresponding ASCII code. For example, if the user enters the numerical expression "2+3", it will be received as the sequence of ASCII codes: 50 (ASCII code for '2'), 43 (ASCII code for '+'), and 51 (ASCII code for '3'). It is important to note that the character value such as '3' is not the same as the numerical value 3. To perform any mathematical calculations, we must convert the character value '3' and ‘2’ into its numerical value and then perform the addition operation which yields the decimal value of 5.

While decimal numbers are used for calculations (e.g., 2+3=5), monitor output requires converting the numerical result back into ASCII codes to display the desired characters on the screen. This is because the monitor interprets and displays characters based on their corresponding ASCII values. If the value of 5 was outputted without any conversions, the monitor would interpret it as the ASCII code for the "enquiry" character (ASCII value 5) instead of the desired character value of 5. To correctly display the character '5' on the monitor, it is necessary to convert the numerical value of 5 to its corresponding ASCII representation. In ASCII, the character '5' is represented by the decimal value 53 or the hexadecimal value 0x35.

By converting the numerical value of 5 to the ASCII code 53 or 0x35, the monitor will display the character '5' as intended.

For this project, I have developed a single integer calculator entirely in assembly language. The calculator takes in 6 possible operations: addition(+), subtraction(-), multiplication(*), division(/), modulus(%), and exponentiation(^). When it comes to x86 assembly, specific instructions are utilized to perform these operations.

Additions uses the **‘add’** instruction to add two values. For example, **‘add rax, rcx’** adds the values in rax and rcx and stores the result in rax.

Subtraction uses the **‘sub’** instruction to subtract one value from another. For example, **‘sub rax, rcx’** subtracts the value in rcx from the value in rax and stores the result in rax.

Multiplication can be done using the **‘mul’** or **‘imul’** instruction. Both instructions multiply the value in rax with another value. The product is stored in rax. If the product is too large to fit in rax, the higher bits are stored in rdx.

Division can be performed using the **‘div’** or **‘idiv’** instruction. These instructions divide the value in rax by another value. The quotient is stored in rax, and the remainder (if any) is stored in rdx.

Modulus is obtained from the division operation. After performing division using **‘div’** or **‘idiv’,** the remainder is stored in rdx.

Exponentiation can be achieved by using a loop and multiple **‘mul’** instructions. You can set up a loop that iterates a specified number of times, using the **‘mul’** instruction to multiply the value in rax by itself in each iteration. The final result will be stored in rax.

Design Principle:

When designing this program, I decided to utilize functions, macros, and for loops to decipher through the user input, evaluate the given expression, and output the answer. Using macros, I outputted the prompt and took in the user input of 14 characters (expression including the enter key).

```
print   msg1, 23
scan    buffer, 14
```

I then converted the first character into a decimal and stored it in the variable 'result'. That way, we are able to initialize 'result', which can then be changed as we iterate through the expression. To convert decimal values into their corresponding ASCII codes, we can use the 'and' instruction. This instruction takes the binary representation of the decimal value and performs a bitwise 'and' operation with 0x0F (represented by the binary value 0000 FFFF). The result of this operation will be the ASCII code of the decimal number.

```
and     byte[buffer], 0fh           ;turns character into decimal value
mov     ah, byte[buffer]
add     byte[result], ah
```

I then iterated through the rest of the user input two characters at a time (the operator and the next number) using a loop. Through each iteration, the next number was converted into a decimal. We then call the ‘sign’ function where the operation character is passed as the first argument(dil), and the decimal number is passed as the second argument(sil). After we save the decimal value in its respective parameter register, we convert it back to ascii in order to print the buffer at the end.

```
next:
        mov     dil, byte[buffer + r10]
        inc     r10
        and     byte[buffer + r10], 0fh
```

```
	mov	sil, byte[buffer + r10]
	add	byte[buffer + r10], 30h		;convert number back to ascii

	call	sign
```

In the function 'sign', we compare the current operation(dil) with all possible hex representations of the operations and jump to its respected label when found. When reaching the correct label, the operation is performed and stored in 'result'.

```
sign:
	cmp	dil, 0x2B
	je	add_sign
	cmp	dil, 0x2D
	je	sub_sign
	cmp	dil, 0x2A
	je	mul_sign
	cmp	dil, 0x2F
	je	div_sign
	cmp	dil, 0x25
	je	mod_sign
	mov	dl, 1
	mov	cl, byte[result]
	cmp	dil, 0x5E
	je	power_sign

	jmp	end

	;addition
	add_sign:
		add	byte[result], sil
		jmp	end

	;subtraction
	sub_sign:
		sub	byte[result], sil
		jmp	end

	;multiplication
	mul_sign:
```

```
        mov     al, byte[result]
        mul     sil
        mov     byte[result], al
        jmp     end

;division
div_sign:
        mov     al, byte[result]
        mov     ah, 0
        idiv    sil
        mov     byte[result], al
        jmp     end

;modulus
mod_sign:
        mov     al, byte[result]
        mov     ah, 0
        idiv    sil
        mov     byte[result], ah
        jmp     end

;to the power
power_sign:
        mov     al, byte[result]
        cmp     dl, sil
        jl      power
        jmp     end
        power:
                inc     dl
                mul     cl
                mov     byte[result], al
                jmp     power_sign
;done
end:
        ret
```

After each iteration, we increase r10 to get to the next operation character, and iterate again if r10 does not equal 13(the number of inputted characters).

```
inc     r10
```

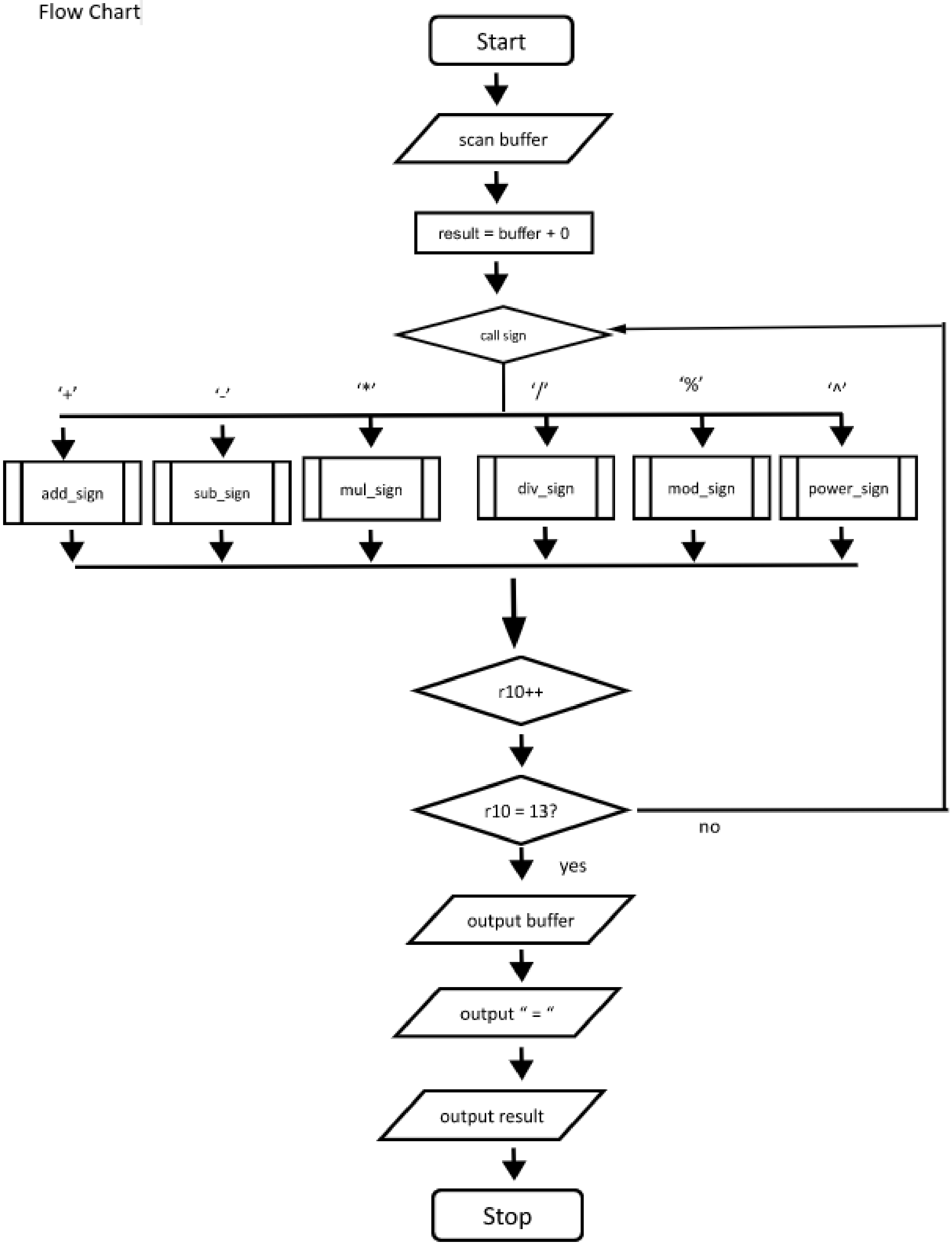
```
cmp     r10, 13
jne     next
```

Once we have completed all the iterations and the calculations, the final result should be stored in the variable 'result'. However, we must convert that result into an ascii value. Since the functionality of this program is limited to outputting a result that is a single digit (ranging from 0 to 9), we can convert the numerical value of 'result' into its ASCII representation without any division. To achieve this, we can add the value of 'result' to the character '0'. In the program, the character '0' is represented by the variable 'ascii' and is stored as the hexadecimal value 0x30. By adding 'result' to 'ascii', we are essentially performing an arithmetic operation between a numerical value and a character value. Since '0' is represented as 0x30 in hexadecimal, adding a single-digit decimal value to it results in the conversion of that decimal value into its corresponding ASCII character. For example, if the value of 'result' is 5, adding it to '0' would yield the ASCII code for the character '5', which is 0x35. By performing this addition, we convert the numerical value of 'result' into its ASCII representation, allowing us to output the calculated result as a character on the monitor. We can now print the buffer, along with the string “ = “ , and ‘result’ in character form.

```
print   buffer, 13
print   msg2, 3
mov     ah, byte[result]
add     byte[ascii], ah
print   ascii, 2
```

Flow Chart and Program Code:

Flow Chart



```
%macro  print   2
        mov     rax, 1                                          ;SYS_write
        mov     rdi, 1                                          ;standard output device
        mov     rsi, %1                                         ;output string address
        mov     rdx, %2                                         ;number of character
        syscall                                                 ;calling system services
%endmacro

%macro  scan    2
        mov     rax, 0                                          ;SYS_read
        mov     rdi, 0                                          ;standard input device
        mov     rsi, %1                                         ;input buffer address
        mov     rdx, %2                                         ;number of character
        syscall                                                 ;calling system services
%endmacro

section .data
        msg1    db      "Enter your expression: "
        msg2    db      " = "
        ascii   db      "0", 10

section .bss
        buffer          resb    14
        result          resb    1

section .text
        global  _start

_start:
        ;prompt and user input
        print   msg1, 23                                        ;using print macro to print msg1
        scan    buffer, 14                                      ;using scan macro to take in user input(the expressio


        ;taking the first number and adding it to result (acts as like a base case)
        and     byte[buffer], 0fh                               ;converting first number into decimal
        mov     ah, byte[buffer]                                ;ah = first number
        add     byte[result], ah                                ;result = result + ah

        ;converting first number back into ascii in order to print it again
        add     byte[buffer], 30h

        ;counter (we already added the first number to result, so we start with 1 instead of 0)
        mov     r10, 1
next:
        mov     dil, byte[buffer + r10]                         ;moving operation to first argument for function
        inc     r10                                             ;r10++
        and     byte[buffer + r10], 0fh                         ;converting next ascii into number
        mov     sil, byte[buffer + r10] ;second num             ;moving next number into 2nd argument for function

        add     byte[buffer + r10], 30h                         ;num + 30h -> converts number back into ascii
```

```
        call    sign                                            ;calling function with arguments above
        inc     r10                                             ;r10++ -> to access next operation
        cmp     r10, 13                                         ;compare r10 and 13(number of characters from user input)
        jne     next                                            ; if r10 != 13, jump to next:

        ;runs when all number have been dealt with
        print   buffer, 13                                      ;using print macro to print buffer which are still all in ascii
        print   msg2, 3                                         ;using print macro to print " = "
        mov     ah, byte[result]                                ;ah = result
        add     byte[ascii], ah                                 ;ascii = ascii + ah = 30h + ah (converts result into ascii)
        print   ascii, 2                                        ;using print macro to print the result

        ;end system call
        mov     rax, 60
        mov     rdi, 0
        syscall
```

```
;*****************FIND THE OPERATION FUNCTION******************************

global sign

sign:

;compares the current operation(dil) with all possible hex representations
;of the operations, and jumps to its respected label when found
        cmp     dil, 0x2B
        je      add_sign
        cmp     dil, 0x2D
        je      sub_sign
        cmp     dil, 0x2A
        je      mul_sign
        cmp     dil, 0x2F
        je      div_sign
        cmp     dil, 0x25
        je      mod_sign
        mov     dl, 1                                   ;dl = 1 -> intializing counter for power_sign
        mov     cl, byte[result]                        ;cl = result; initializing cl for power_sign
        cmp     dil, 0x5E
        je      power_sign

        jmp     end
```

```

        ;addition
        add_sign:
                add     byte[result], sil
                jmp     end

        ;subtraction
        sub_sign:
                sub     byte[result], sil
                jmp     end

        ;multiplication
        mul_sign:
                mov     al, byte[result]
                mul     sil
                mov     byte[result], al
                jmp     end

        ;division
        div_sign:
                mov     al, byte[result]
                mov     ah, 0
                idiv    sil
                mov     byte[result], al
                jmp     end

        ;modulous
        mod_sign:
                mov     al, byte[result]
                mov     ah, 0
                idiv    sil
                mov     byte[result], ah
                jmp     end
```

```
        ;to the power
        power_sign:
                mov     al, byte[result]
                cmp     dl, sil
                jl      power
                jmp     end
                power:
                        inc     dl
                        mul     cl
                        mov     byte[result], al
                        jmp     power_sign
        ;done
        end:
                ret
```

Simulation Results:

```
andrewss@andrewss-ThinkPad-T480:~/CPSC_240/Assignments/final_project$ ./calc2
Enter your expression: 9+9+9+9+9+9/9
9+9+9+9+9+9/9 = 6
andrewss@andrewss-ThinkPad-T480:~/CPSC_240/Assignments/final_project$ ./calc2
Enter your expression: 4+8/3^4-9%2*5
4+8/3^4-9%2*5 = 5
andrewss@andrewss-ThinkPad-T480:~/CPSC_240/Assignments/final_project$ ./calc2
Enter your expression: 8*7/2+9-3%8^2
8*7/2+9-3%8^2 = 4
andrewss@andrewss-ThinkPad-T480:~/CPSC_240/Assignments/final_project$ ./calc2
Enter your expression: 1+2+3-4+5-6+8
1+2+3-4+5-6+8 = 9
andrewss@andrewss-ThinkPad-T480:~/CPSC_240/Assignments/final_project$ ./calc2
Enter your expression: 9*9+9-0/1%9^2
9*9+9-0/1%9^2 = 0
```

Enter your expression: 4+8/3^4-9%2*5

result	dil	r10++	sil	sign	result	r10++	cmp r10, 13
4	'+'	r10 = 2	8	jmp add_sign	12	r10 = 3	r10 ≠ 13 jump
12	'/'	r10 = 4	3	jmp div-sign	4	r10 = 5	r10 ≠ 13 jump
4	'^'	r10 = 6	4	jmp power_sign	256	r10 = 7	r10 ≠ 13 jump
256	'-'	r10 = 8	9	jmp sub-sign	247	r10 = 9	r10 ≠ 13 jump
247	'%'	r10 = 10	2	jmp mod-sign	1	r10 = 11	r10 ≠ 13 jump
1	'*'	r10 = 12	5	jmp mul_sign	5	r10 = 13	r10 = 13 stop

result = 5

As seen in the chart, we can interpret the final value of 'result' for the expression "4+8/3^4-9%2*5" by iterating through each character, converting its numerical value, interpreting the corresponding operation, and performing the mathematical operation. The result is then updated in the 'result' variable.

Conclusion:

This project successfully implemented a one-integer calculator program in assembly language. By understanding the differences of ASCII codes and decimal numbers, we were able to convert character input into numerical values and perform mathematical operations. This program reinforced the importance of functions and macros to utilize memory space as well as provide cleaner and easy-to-read code. Through simulations and diagrams, we were able to provide step-by-step instructions on how the program is run, as well as verify the accuracy of the results. Overall, this project reinforced my understanding of the intricacies of x86 assembly language.