Single-Digit Integer Calculator

Ryan Nishikawa
885486761
5/10/2024

California State University, Fullerton
Department of Computer Science

I. Introduction

A. Keyboard Input (ASCII code) vs Decimal Number

1. Keyboard input is typically represented with ASCII code and is the process of entering data through a keyboard. ASCII, the American Standard Code for Information Interchange, assigns a unique code to each character. Below is a chart that shows which code represents each character.

Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value	Hex	Value
00	NUL	10	DLE	20	SP	30	0	40	@	50	P	60	`	70	p
01	SOH	11	DC1	21	!	31	1	41	A	51	Q	61	a	71	q
02	STX	12	DC2	22	"	32	2	42	B	52	R	62	b	72	r
03	ETX	13	DC3	23	#	33	3	43	C	53	S	63	c	73	s
04	EOT	14	DC4	24	$	34	4	44	D	54	T	64	d	74	t
05	ENQ	15	NAK	25	%	35	5	45	E	55	U	65	e	75	u
06	ACK	16	SYN	26	&	36	6	46	F	56	V	66	f	76	v
07	BEL	17	ETB	27	'	37	7	47	G	57	W	67	g	77	w
08	BS	18	CAN	28	(	38	8	48	H	58	X	68	h	78	x
09	HT	19	EM	29	)	39	9	49	I	59	Y	69	i	79	y
0A	LF	1A	SUB	2A	*	3A	:	4A	J	5A	Z	6A	j	7A	z
0B	VT	1B	ESC	2B	+	3B	;	4B	K	5B	[	6B	k	7B	{
0C	FF	1C	FS	2C	,	3C	<	4C	L	5C	\	6C	l	7C	|
0D	CR	1D	GS	2D	-	3D	=	4D	M	5D	]	6D	m	7D	}
0E	SO	1E	RS	2E	.	3E	>	4E	N	5E	^	6E	n	7E	~
0F	SI	1F	US	2F	/	3F	?	4F	O	5F	_	6F	o	7F	DEL

Decimal numbers use the first 10 digits (0-9) to represent values, and we can use these values for things like arithmetic.

B. Decimal Number vs Monitor Output (ASCII code)

1. Decimal numbers here are the same as they are right above this. They're used to store values in the computer and in Assembly we normally use decimal for basic arithmetic or as a counter. Monitor output uses the same ASCII code as keyboard input. Still, the difference is while keyboard input is

the encoding of characters, monitor output is the visual representation of the characters.

C. <u>Four Basic Arithmetic Operations in the Assembly Language</u>

1. **Addition**
    a) Adding the values of 2 numbers together.
    b) The general form of integer addition is:
        (1) add <dest>, <src>
        (2) C/C++ equivalent would be <dest>=<dest>+<src> or <dest> += <src>
    c) The value of the source operand stays the same while the result is stored in the destination operand overwriting the value previously there.
    d) For this to work, the destination and source must be of the same size (ex. Both bytes, word, dword).
2. **Subtraction**
    a) Subtracting the value of 1 number from another.
    b) The general form of integer subtraction is:
        (1) sub <dest>, <src>
        (2) C/C++ equivalent would be <dest>=<dest>-<src> or <dest> -= <src>
    c) The value of the source operand stays the same while the result is stored in the destination operand overwriting the value previously there.
    d) For this to work, the destination and source must be of the same size (ex. Both bytes, word, dword).
3. **Multiplication**
    a) Multiplying the value of one number by the value of another number and is split into 2 types, signed and unsigned
    b) The general form of **unsigned** integer multiplication is:
        (1) mul <src>
        (2) C/C++ equivalent would be <A>=<A>*<src> or <A> *= <src>
    c) For multiplication, the A register (rax, eax, ax, ah, al) must be used for one of the operands while the other can be a memory location or register

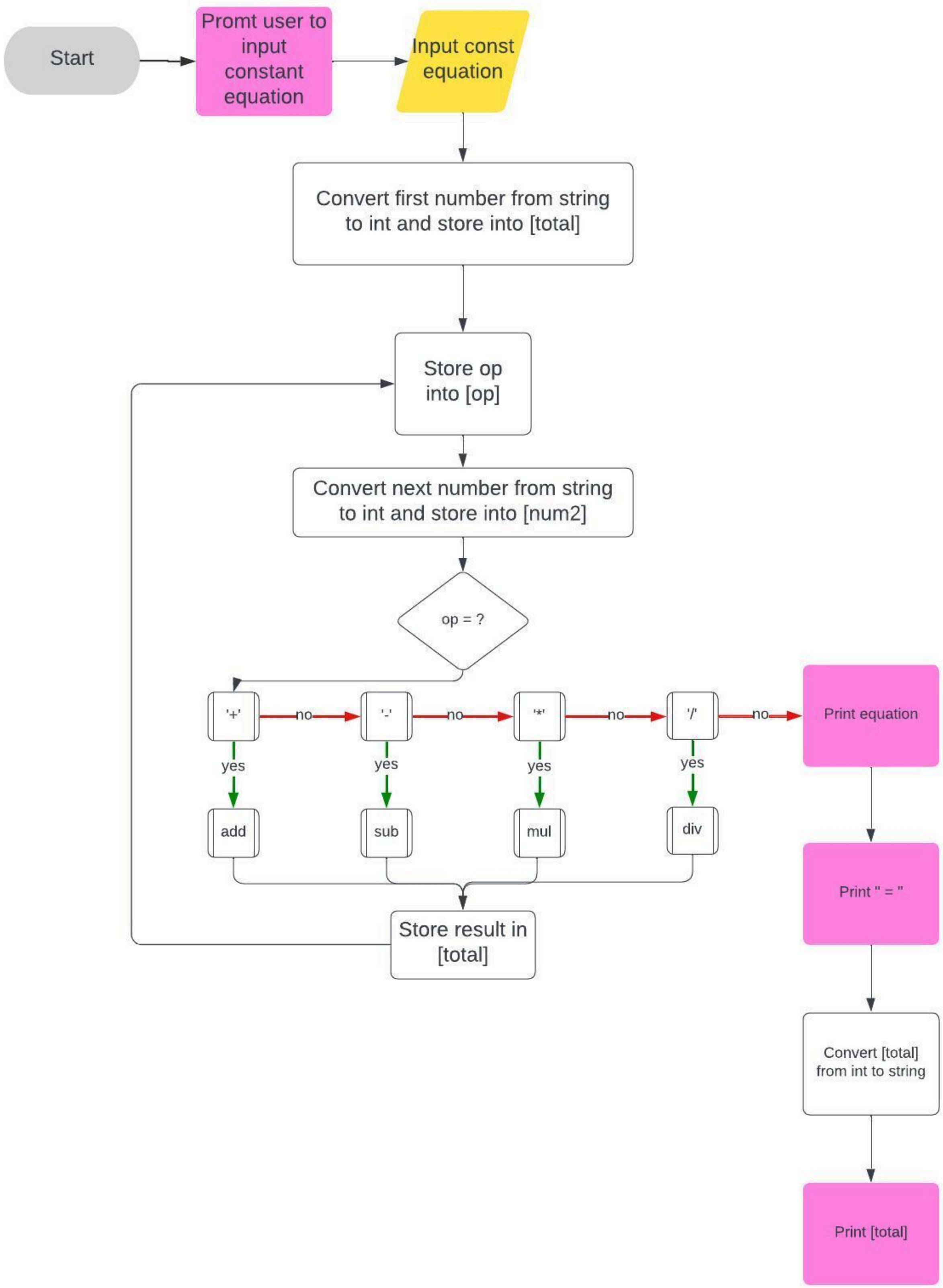
- d) The result will be stored in the A register, but if it’s too big it could be split and stored in the D register as well
- e) The general forms of **signed** integer multiplication are:
    - (1) imul <src>|| imul <dest>, <src/imm> ||
      imul <dest>, <src>, <imm>
    - (2) C/C++ equivalents would be <A>=<A>*<src>, <dest>=<dest>*<src>, or <dest>=<src>*<imm>
- f) The size of the **imm**ediate value is limited to the size of the source operand and goes up to 32 bits.
- g) The result is truncated to the size of the destination operand, however a byte sized destination is not supported

4. Division

- a) Dividing the value of one number by the value of another number and is split into 2 types, signed and unsigned
- b) The general form of integer division is:
    - (1) div <src> ;unsigned
      idiv <src ;signed
    - (2) C/C++ equivalents would be <A>=<A>/<src>
    - (3) For division, the A register (rax, eax, ax, ah, al) and possibly the D register (rdx, edx, dx) must be used for the dividend
    - (4) The divisor can be a register or memory location but not an immediate and the result will be stored in one of the A registers while the remainder will be stored in ah, dx, edx, or rdx depending on the size.
    - (5) Do not divide by zero as doing this will crash the program and make the moon collide with Earth.
      ||
      ||
      ||
      ||
      ||
      \/

II. Design Principle (Algorithm)

A. Flow Chart



B. Assembly program

```
%macro  print   2
    mov     rax, 1                  ;SYS_write
    mov     rdi, 1                  ;where to write
    mov     rsi, %1                 ;address of strint
    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 .bss
buffer    resb 63
bufferLen resb 1
op        resb 1
num2      resb 1
total     resb 1
ascii     resb 4
asciiLen  resb 1

section .data
LF              equ 10
NULL            equ 0
SYS_exit        equ 60
EXIT_SUCCESS    equ 0
msg1            db "Enter Operations String: ", NULL
msg2            db " = ", NULL

section .text
    global _start
_start:
    print   msg1, 25
    scan    buffer, 63                  ;get equation (to change max size change .bss and this)
    mov     r10, 0                      ;initialize counter
    mov     al, byte[buffer + r10]
    and     al, 0fh                     ;total = atoi(buffer[r10])
    mov     byte[total], al             ;get first num of set
    inc     r10
calcLoop:
    mov     al, byte[buffer + r10]
    mov     byte[op], al                ;get op of set
    inc     r10

    mov     al, byte[buffer + r10]      ;get second num of set
    and     al, 0fh                     ;num2 = atoi(buffer[r10])
    mov     byte[num2], al
    inc     r10
checkAdd:
    cmp     byte[op], '+'               ;if op is '+'
    jne     checkSub                    ;[total] += num2
    mov     dil, byte[total]
    mov     sil, byte[num2]
    call    addition
    jmp     calcLoop
```

```
checkSub:
    cmp     byte[op], '-'                   ;if op is '-'
    jne     checkMult                       ;[total] -= num2
    mov     dil, byte[total]
    mov     sil, byte[num2]
    call    subtract
    jmp     calcLoop
checkMult:
    cmp     byte[op], '*'                   ;if op is '*'
    jne     checkDiv                        ;[total] *= num2
    mov     dil, byte[total]
    mov     sil, byte[num2]
    call    mult
    jmp     calcLoop
checkDiv:
    cmp     byte[op], '/'                   ;if op is '/'
    jne     next1                           ;[total] /= num2
    mov     dil, byte[total]                ;if op is not divide we reached the end of the equation
    mov     sil, byte[num2]
    call    divide
    jmp     calcLoop
next1:
    dec     r10                             ;dec r10 to match the size of eqn
    dec     r10
    print   buffer, r10                     ;print the buffer without null chars
    print   msg2, 3                         ;print the " = "
    call    toString                        ;ascii = itoa(total)
    print   ascii, [asciiLen]               ;print the total without null chars
;end
    mov     rax, SYS_exit                   ;terminate excuting process
    mov     rdi, EXIT_SUCCESS               ;exit status
    syscall


addition:
    mov al, dil
    add al, sil
    mov byte[total], al
    ret
```

```
subtract:
    mov al, dil
    sub al, sil
    mov byte[total], al
    ret
mult:
    mov al, dil
    mul sil
    mov byte[total], al
    ret
divide:
    mov al, dil
    div sil
    mov byte[total], al
    ret
```

```
;ascii = itoa(total); convert total from int to string
toString:
    ; Part A - Successive division
        mov     al, byte[total]
        mov     rcx, 0
        mov     rbx, 10
    divideLoop:
        mov     edx, 0
        div     rbx
        push    dx
        inc     rcx
        cmp     eax, 0
        jne     divideLoop
        inc     cl
        mov     byte[asciiLen], cl
        dec     cl

    ; Part B - Convert remainders and store
        mov     rbx, ascii
        mov     rdi, 0
    popLoop:
        pop     ax
        add     al, "0"
        mov     byte [rbx+rdi], al
        inc     rdi
        loop    popLoop
        mov     byte[rbx+rdi], LF
        ret
```

C. <u>Conversion Principle (algorithm) of keyboard input to decimal numbers and symbols</u>
    1. Conversion to decimal numbers in this code is straightforward. Because we only have single-digit numbers, it can be done in one line:
        a) and al, 0fh   ;al is where the input to convert is stored
    2. This can be seen in lines 41 & 50 of the above code

D. <u>Conversion Principle (algorithm) of decimal number to ASCII code</u>
    1. Conversion to ASCII code from decimal number is also very straightforward and con be done in one line:
        a) add al, “0”   ;al is where the num to convert is stored
    2. However, the rules of the project only say the input numbers should be single digits, not the output. To account for an

output that could be multiple digits long we must convert it one digit at a time. This can be done by:

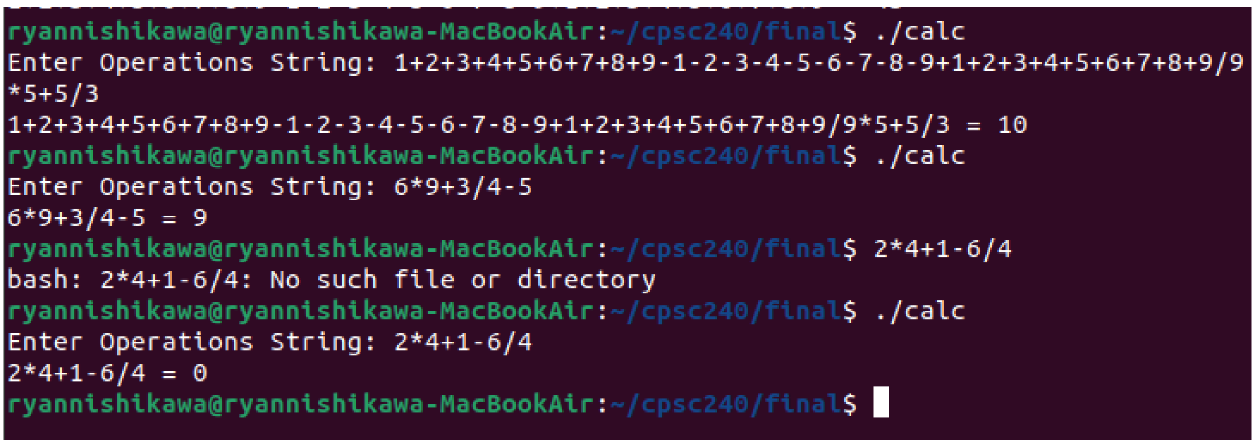
a) divideLoop – divide the number by 10 then store the remainder (repeat until the number is 0)

b) popLoop – convert the numbers to ASCII one by one and store them in a string

3. This can be seen in the toString function at the end of the code

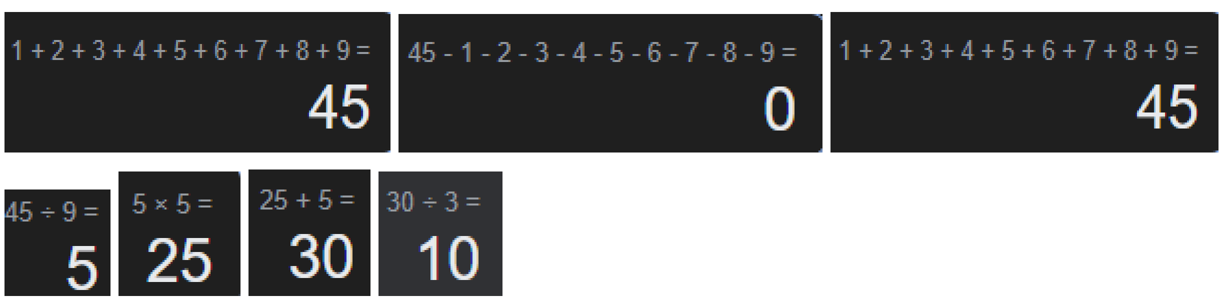
III. Simulation Results

A.



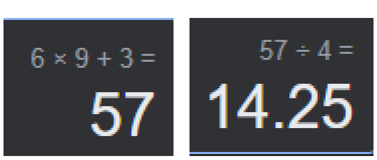
```
ryannishikawa@ryannishikawa-MacBookAir:~/cpsc240/final$ ./calc
Enter Operations String: 1+2+3+4+5+6+7+8+9-1-2-3-4-5-6-7-8-9+1+2+3+4+5+6+7+8+9/9
*5+5/3
1+2+3+4+5+6+7+8+9-1-2-3-4-5-6-7-8-9+1+2+3+4+5+6+7+8+9/9*5+5/3 = 10
ryannishikawa@ryannishikawa-MacBookAir:~/cpsc240/final$ ./calc
Enter Operations String: 6*9+3/4-5
6*9+3/4-5 = 9
ryannishikawa@ryannishikawa-MacBookAir:~/cpsc240/final$ 2*4+1-6/4
bash: 2*4+1-6/4: No such file or directory
ryannishikawa@ryannishikawa-MacBookAir:~/cpsc240/final$ ./calc
Enter Operations String: 2*4+1-6/4
2*4+1-6/4 = 0
ryannishikawa@ryannishikawa-MacBookAir:~/cpsc240/final$
```

B. Results for 1

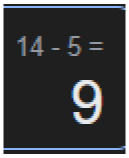


Therefore,
1+2+3+4+5+6+7+8+9-1-2-3-4-5-6-7-8-9+1+2+3+4+5+6+7+8+9/9*5+5/3 = 10 when going from left to right

C. Results for 2

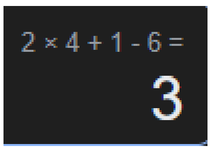


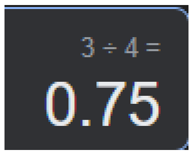
However, because we are using integers, not double/float numbers, we round 14.25 down to 14



Therefore, 6*9+3/4-5 = 9 when going from left to right

D. Results for 3





However, because we are using integers, not double/float numbers, we round 0.75 down to 0

E. This was verified using the Google calculator

IV. Conclusion

A. In conclusion, developing this calculator was both challenging and rewarding. Many parts were easy like the basic arithmetic functions and coming up with the algorithm. What I struggled with the most was registers. Although I've changed many programs to use higher registers, this was the first time I had to change programs to use lower registers. More specifically, I struggled with getting the buffer length. At first, I was trying to use RCX as a counter because I had done that in previous assignments, however, no matter what I did it threw segmentation faults everywhere. After looking through more old assignments, I decided to use r10 as a counter instead, which fixed everything. Another problem I had was converting from decimal to string. Similar to the counter I had converted the function to deal with higher bits many times, but this time I couldn't figure out which registers to lower. Through my research trying to figure this out, I feel like I've learned a lot more about registers and I feel a lot more confident when dealing with the Assembly language.