CS 271 Homework 3

Accumulators and Integer Arithmetic

Due Date: 07/28/19

Michael Payne

TA Signature

1 Introduction

Homework 3 gets the user's name and then asks them to enter a series of negative integers. It then sums the integers up and gets their average. Having to extensively deal with negative numbers was a relatively new thing, but other than that, homework 3 was mostly about practicing working in assembly and reinforcing concepts that we had already covered. The most challenging part of the assignment was simply trying to deal with negative integers. The additional requirements of IDIV and IMUL did not mix well with the method that I used to generate the three digits after the decimal place (I will elaborate on this in later sections).

2 Program Initialization, Internal Register, Definitions, and Constants

In the .data section, "greeting1", "question1", "hello", "goodbye1", "goodbye2", "period", "inst1", "inst2", "numLine1", "numLine2", "result1", "result2", "result3", "result4", "programTime", "noIntegers", and "errorNum" are all strings that are given a byte for each character. A string named "userName" is assigned 33 bytes. "numnum" and "upperLimit" are each assigned 4 bytes and they both have values of 0. "tenten" is assigned 4 bytes and has a value of 10. "numTotal" and "timeStamp" are assigned 4 bytes and are both uninitialized. "lowerLimit" is assigned 4 bytes and has a value of -100.

3 Main Program

Figure 1 shows all of the main components and functional logic of the program. The "Introducing Program and Greeting User" segment starts by printing the title of the program and my name. It then gets the user's name and stores it in "userName". The program greets the user and then moves onto the "Program Instructions and User Num Input" segment, which starts by printing instructions that tell the user to input an integer within the domain of -100 to -1. It also says that when the user wants to stop inputting numbers that all they need to do is input a number greater than -1 (this is done in a block of code labeled "inputStart"). The program clears out eax and progresses to a block of code labeled "printLine", which is a loop that first adds whatever is in eax to a location in memory labeled "printLine" (this will be 0 at first). The program then calls a procedure called "numberLine". This procedure increments "numnum" (the count of correcty inputted integers), and then prints a numbered input line (the number is determined by "numnum"). Once the user inputs something, the procedure jumps back to main (back into "printLine") and runs the inputted integer through a series of checks. If the integer is less than -100, the program jumps up to a block of code labeled "errorMSG", which prints an error message and decrements "numnum" (the program then moves through the "inputStart" code again). The next check looks to see if the integer is less than 0. If it is, the program jumps back to the beginning of "printLine", and the input process starts all over again. If the user inputs an integer greater than 0, the program decrements "numnum" (to essentially ignore the inputted non-negative integer). It then checks the negative integer count ("numnum"). If it is 0, the program jumps to "zeroIntegers" where a special message is printed (about there being nothing to calculate), and the program runs through the final 2 concluding segments. If the user did actually enter some negative integers, the program moves on to the "Calculating and Printing Results" segment, which calls the “printResults” procedure. This procedure first prints the number of inputted negative integers. It then prints the sum of those integers. Finally, it calculates the average of the summed negative integers (by dividing the sum by the total number of entered integers). The digits to the left of the decimal place are generated and printed in “printResults”, and the 3 digits to the right are generated and printed by a procedure called “divisionLoop”. The program returns to main and jumps to a segment called “Saying Goodbye”, which does exactly that. It then moves on to the final segment, which is called “Printing Program Execution Time”. The program actually takes a timestamp at the very beginning with the rdtsc instruction and stores the lower half of the generated number into memory labeled “programTime”. The program takes another timestamp at the very end and then subtracts the value in “programTime” from it. Finally, it prints this number and the program ends.

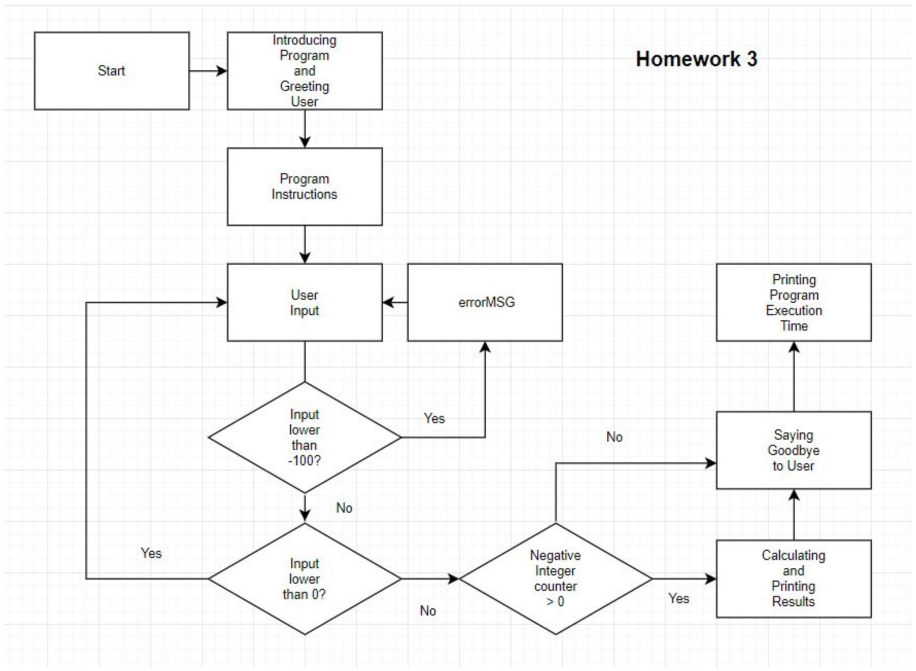

Figure 1: Block Diagram for Homework 3

4 Procedures

1. “numberLine”

   “numberLine” is called by “printLine” in order to print integer input lines for the user. It increments the integer count (“numnum”) and then prints a numbered integer input line (a combination of the number from “numnum”, and the strings “numLine1” and “numLine2”).

2. “printResults”

   This procedure is called at the beginning of the “Calculating and Printing Results” segment. It prints out the total number of inputted negative integers, the sum of those integers, and then, finally, the average. The first half of the average (the part to the left of the decimal point) is calculated and printed in “printResults”. The procedure “divisionLoop” handles the digits to the right of the decimal point.

3. “divisionLoop”

   “divisionLoop” takes the remainder of the division of the sum of the inputted integers divided by the number of inputted integers and then uses that to generate the 3 digits to the right of the decimal point (in the average that is being calculated) . It multiplies the remainder by 10 and divides it by “numnum”. The result is then printed. This process is repeated 2 more times (the newly generated remainders are used for each subsequent digit).

5 Stored Program Data

1. “Introducing Program and Greeting User” stage
   (a) greeting1 = “Welcome to the integer adder by Michael Payne”
   (b) question1 = “What is your name: ”
   (c) hello = “Hello, ”
   (d) goodbye1 = “Thanks for using my program.”
   (e) goodbye2 = “Goodbye, ”
   (f) UserName = 33 DUP(0) = is changed to “Mike”
   (g) period = “.”
   (h) inst1 = “Please enter numbers in [-100, -1].”
   (i) inst2 = “Enter a non-negative number when you are finished.”
   (j) numLine1 = “: ”
   (k) numLine2 = “Enter number: ”
   (l) result1 = “You entered “
   (m) result2 = “ negative number(s).”
   (n) result3 = “The sum of your numbers is ”
   (o) result4 = “The rounded average is -”
   (p) programTime = “The elapsed program time is: ”
   (q) noIntegers = “You entered zero integers.”
   (r) errorNum = “You cannot enter a number less than -100!”
   (s) numnum = 0
   (t) tenten = 10
   (u) numTotal = ?
   (v) upperLimit = 0
   (w) lowerLimit = -100
   (x) timeStamp = ? = some random time stamp that is taken from eax (which is generated by rdtsc)
2. “Program instructions and User Num Input” stage

    (a) numnum = increments from 0 to 4 and then decrements to 3 at the end of the segment
    (b) numTotal = negative integers of -30, -30, and then -40 are summed to -100

3. "Calculating and Printing Results" stage
    (a) numnum = increments from 0 to 4 and then decrements to 3 at the end of the segment numTotal = -100

4. "Saying Goodbye" stage
    (a) Nothing changed

5. "Printing Program Execution Time" stage
    (a) timeStamp = new timestamp is generated and old one is subtracted from it; let's say it is 10000

6 Difficulties and Concluding Thoughts

I tried to take a more modular approach with this assignment than I did with the first 2 assignments. While planning the assignment out, I divided it into different modules, programmed them separately, and then linked them all together into one asm file. This isn't groundbreaking stuff (people approach problems with any degree of complexity in this manner pretty regularly), but it was quite a bit different than how I had previously programmed things in this class. This made debugging quite a bit more efficient, because instead having a huge block of code that I progressively added to and tested at arbitrary points, I had a bunch of separate functional segments that I had already thoroughly tested. This resulted in my program having a lot of procedures, because that seemed to be the easiest way to link everything together.

However, while my program does everything that the assignment asks for, I feel like the overall structure could definitely be improved upon as I didn't put enough thought into how I was going to link everything together when I was planning my assignment out. My belief is that if you're going to put something outside of the main block of code and into a procedure, it should be because it makes the program run more quickly and efficiently (at least in assembly) – not because you're cobbling together a bunch of disconnected pieces of code.

As for the individual challenges or tasks in the assignment, everything was fairly straightforward. The most difficult part (which I mentioned in the introduction) was dealing arithmetic with negative integers, because I wasn't terribly familiar with IMUL or IDIV (I'll elaborate on this in the extra credit section). This was primarily challenging, because IDIV did not mesh well with some of the things I did to complete the extra credit portions of the assignment (I will elaborate on that in the extra credit section).

7 Extra Credit

This program successfully completes all of the extra credit tasks in this assignment.

Numbering the input lines was fairly easy . Initially, I tried to overcomplicate the solution by incrementing a hex number and using the ascii table, but I realized that it would be far easier to simply use the integer input counter ("numnum"). The "numberLine" procedure used the aforementioned counter to attach a number to each printed input line.

And tracking the execution time of the program was fairly simple as well. I took a timestamp at the beginning of the program with rdtsc, stored the value in a memory location, and then subtracted that value from another timestamp that was taken at the end of the program.

The final part of the extra credit was by far the most challenging (for a number of reasons) of the 3. I made the mistake of actually trying to use real floating point numbers at first, and I spent a lot of time trying to make that work. I eventually realized how insanely difficult it would be to print a floating point number in the form that the assignment wanted, so I gave up on that idea and settled on generating all of the numbers to the right of the decimal point by dividing the remainder (of the original division) by the integer count. For example, if you divide 10 by 7, you're going to get 1 with a remainder of 3. Simply multiply the 3 by 10 and then divide that by 7 again. That will give you a 4 with a remainder of 2. Multiply the 2 by 10 and divide that by 7 and you have a 2 with a remainder of 6. Multiply the 6 by 10 and divide by 7 and you have an 8. If you combine all of the digits into 1 number (with the decimal in the appropriate place), you get 1.428, which is the proper result of 10 divided by 7 in decimal form with the rounding that the assignment asks for. It later dawned on me (while I was going through my code and beefing up the comments) that I simply could have multiplied the remainder by 1000 and then divided by the integer count, and that would have given me the 3 digits that I needed, but I opted to use my other (albeit more complicated and obnoxious) solution, because I didn't want to have to spend time coding the conditionals to deal with some of the weird situations where 3 numbers weren't generated (I guess specifically where the remainder was 0).

Also, since I was using signed integers, I couldn't use regular DIV and MUL. This was problematic, because IDIV didn't produce a remainder in EDX that seemed usable to me. Instead of trying to wrangle with IDIV, I decided to get the absolute value of the sum of the negative integers (I just stuck a '-' character in front of my average when printing out the result), so that I could use regular div. This was far more compatible with the method that I used to generate a floating point number.

8 Source Code

```
TITLE Homework3 (Michael_Payne_hw3_code.asm)

; Author: Michael Payne
; Last Modified: 07/27/2019
; OSU email address: paynemi
; Course number/section: cs271
; Project Number: 3 Due Date: 07/28/2019
; Description: This program sums up negative integers inputted by the
   user. It first introduces the program title and my nanme.
; It then asks the user his/her name and then greets them (with their
   name). After that, it tells the user to enter an integer in
; domain of [-100, -1]. Integers less than -100 produce an error message.
    Integers greater than -1 move the program to the results stage.
```

```
; All of the negative numbers (number that ends input stage is discarded)
    are summed up, and then the sum is divided by the number
; of inputted integers to give an average. The program rounds up to the
   nearest .001. The program then says goodbye to the user.
; Also, the program uses the rdtsc instruction to keep track of how much
   time has elapsed from the beginning of the program to the end.
; It prints the results at the end after saying goodbye.

INCLUDE Irvine32.inc

.data
greeting1  BYTE "Welcome to the integer adder by Michael Payne.",10
question1  BYTE "What is your name: ", 0
hello   BYTE "Hello, ",0
goodbye1  BYTE "Thanks for using my program.",10
goodbye2  BYTE "Goodbye, ",0
userName  BYTE 33 DUP(0)   ; user name
period   BYTE ".",0
inst1   BYTE "Please enter numbers in [-100, -1].", 10
inst2   BYTE "Enter a non-negative number when you are finished to see
   results.", 0
numLine1  BYTE ": "
numLine2  BYTE "Enter number: ",0
result1   BYTE "You entered ",0
result2   BYTE " negative number(s).", 10
result3   BYTE "The sum of your numbers is ", 0
result4   BYTE "The rounded average is -",0
programTime  BYTE "The elapsed program time is: ",0
noIntegers  BYTE "You entered zero integers.",0
errorNum  BYTE "You cannot enter a number less than -100!", 0
numnum   DWORD 0     ; count for number of integers user enters
tenten   DWORD 10     ; need number 10 for calculation of digits after
   decimal point
numTotal  DWORD ?     ; sum of all inputted integers
upperLimit  DWORD 0     ; Used to check to see if number is negative or
   not
lowerLimit  DWORD -100    ; Inputted number cannot be less than -100
timeStamp  DWORD ?


.code
main PROC
; segment 1 - Introducing Program and Greeting User - This segment starts
    the timer for program execution time by generating a timestamp
; with the rdtsc instruction. Later on, this timestamp is used to
   calculate how long it took for the program to execute.
```

```
; This segment also lists the program title and my name. It then gets the
    user name, greets the user with his/her name and then
; jumps to inputStart where program instructions are printed and user
   input is taken and printed
rdtsc         ; getting timestamp of beginning of program
mov   timeStamp, eax    ; saving timestamp so that elapsed time can be
   calculated at end of program
mov   edx, offset greeting1
call  writestring
mov   edx, OFFSET userName  ; using readstring to store user's input of
   name
mov   ecx, 32      ; specifying size of 32 bytes
call  ReadString
mov   edx, offset hello   ; printing greeting to user (with user's name)
call  writestring
mov   edx, offset userName
call  writestring
mov   edx, offset period
call  writestring
jmp   inputStart     ; jumping to inputStart (need to skip errorMSG)

; segment 2 - Program instructions and User Num Input - This segment
   instructs the user to enter an integer that falls within the domain
; of [-100, -1]. It then takes input from the user. If the user enters an
    integer less than -100, the program jumps back to errorMSG,
; and instructs the user to enter an integer greater than or equal to
   -100. If the user enters an integer within the stated domain, the
   program
; adds it to numTotal, and increments numnum (which keeps track of how
   many correctly inputted integers there are; this is used to
; print the numbers for the input lines, and to later produce an average)
   . The procedure numberLine is used to print inputLines.
; If the user inputs an integer greater than -1, the program moves on to
   segment 3. If the user inputs an integer greater than -1 while numnum
; equals 0 (meaning the user hasn't inputted any negative integers), the
   program jumps to zeroIntegers, which prints a special message before
   ending
; the program
; errorMSG is skipped initially, but the program jumps to it if the user
   enters an integer less than -100. It prints an error message
; and then decrements numnum, so the input lines are still correctly
   numbered and the number of correctly entered negative integers
; remains accurate
errorMSG:
mov   edx, offset errorNum  ; printing error message
call  writestring
```

```
dec   numnum      ; decrementing numnum so integer count remains accurate
; inputStart prints instructions for the user, and clears out eax, so
   that numTotal can be properly added to later on
inputStart:
call  crlf
mov   edx, offset inst1   ; printing instructions about user inputs
call  writestring
call  crlf
xor   eax, eax     ; clearing eax so numTotal stays accurate
; printLine is a loop that handles everything related to the user's
   integer inputs. It first adds correct inputs to numTotal (which
; is 0 during the first iteration of the loop). It then calls the
   procedure numberLine (which is explained in detail later), which
; prints out an input line (with the number of the line displayed. the
   number only increments if the inputted number is greater than
; -101). Inputs generated in numberLine are checked against the
   lowerLimit (-100) and upperLimit (0). Inputs less than lowerLimit
   cause
; the program to jump up to errorMSG. Inputs less than upperLimit restart
    the loop. If the user enters a non-negative integer, the loop
; ends, numnum is decremented (to throw out the non-negative integer
   essentially), and the program moves onto the next segment.
; Also, the value of numnum is compared to 0. If it equals zero, the
   program jumps to zeroInteger where a special message is printed
; and the program ends (no point in running through all of stuff in
   segment 3 if the user hasn't entered any negative integers)
printLine:
add   numTotal, eax    ; adding correctly inputted integer to numTotal
call  numberLine     ; calling procedure numberLine to print out numbered
    input lines
call  readint      ; reading user input
cmp   eax, lowerLimit    ; comparing inputted integer to -100
jl   errorMSG     ; if less than -100, jump to errorMSG to scold user
cmp   eax, upperLimit    ; comparing inputted integer to 0
jl   printLine     ; if less than 0, restart loop.
dec   numnum      ; once user has inputted positive integer, program
   moves on and decrements numnum
cmp   numnum,0     ; making sure the user has inputted at least 1
   negative integer
je   zeroIntegers    ; if not, jump to zeroIntegers

; segment 3 - Calculating and Printing Results - Segment 3 calls
   printResults, which takes the negative integers that the user inputted
; and prints out the sum and then the average of the integers (
   description of the procedure will give a more detailed explanation of
   how
; this works exactly). The program then jumps to next segment to say
   goodbye to user and print program execution time
call  printResults    ; calling printResults procedure that processes
   inputted negative integers
jmp   farewell     ; once finished with that, jump to farewell to say
   goodbye to user and print program execution time
; zeroIntegers handles special case where user inputs a non-negative
   integer without having first inputted any negative integers.
; It lets the user know that they haven't entered anything to process or
   calculate and then moves onto next segment of program.
zeroIntegers:
mov   edx, offset noIntegers  ; printing message about no negative
   integers
call  writestring

; segment 4 - Saying Goodbye - This segment thanks the user for using my
   program and then says goodbye (with their name)
farewell:
call  crlf
mov   edx, offset goodbye1  ; saying goodbye and thanking user
call  writestring
mov   edx, offset userName
call  writestring
mov   edx, offset period
call  writestring

; segment 5 - Printing Program Execution Time - this segment prints the
   program execution time. It generates a new timestamp and then
; subtracts the timestamp (the part of the number stored in eax) from it.
    It prints the result (which is the number of elapsed ticks)
call  crlf
mov   edx, offset programTime  ; starting to print elapsed execution time
call  writestring
rdtsc         ; getting a new timestamp
sub   eax,timeStamp ; subtracting old timestamp from that
call  writedec     ; finish printing elapsed execution time

exit ; exit to operating system
main ENDP

; NumberLine is a procedure that prints out input lines for the user. It
   increments numnum and then uses the number stored in numnnum
; to number each input line (numnum is essentially only incremented when
   the user inputs an integer greater than -101).
numberLine PROC
inc   numnum      ; incrementing numnum to keep track of correctly
```

```
   inputted integers
mov   eax, numnum     ; printing number stored in numnum (no line break
   between this and following printed string)
call  writedec
mov   edx, offset numline1  ; printing "Enter Number: "
call  writestring
ret          ; return to main
numberLine ENDP

; The printResults procedure prints out the number of correctly inputted
   negative integers and the sum of those integers. It also gets the
; absolute value of the sum of the negative integers, because when I was
   generating the average of the inputted integers, I needed
; to use the resulting remainder generated from div to produce the 3
   digits after the decimal (in order to print a floating point
; number). idiv was too awkward to use, so I decided to work with non-
   signed numbers and then simply insert a '-' character
; in front of my average. Once it gets the absolute value of sumTotal, it
    divides the sum by the number in numnum to get the
; non decimal digits (the stuff to the left of the decimal point) of the
   average. It then calls division loop, which handles the work
; of generating the three digits to the right of the decimal point
printResults PROC
mov   edx, offset result1   ; printing number of inputted negative
   integers
call  writestring
mov   eax, numnum
call  writedec
mov   edx, offset result2   ; printing sum of inputted negative integers
call  writestring
mov   eax, numTotal
call  writeint
call  crlf
mov   edx, offset result4   ; starting process of printing average of
   inputted negative integers
call  writestring
xor   edx, edx     ; clearing edx for division
imul  eax, -1      ; getting absolute value of summed integers
div   numnum      ; dividing sum of integers by numnum to get average
call  writedec     ; printing digits to left of decimal point
xchg  ebx, edx     ; moving remainder to ebx so decimal point can be
   printed
mov   edx, offset period   ; printing decimal point
call  writestring
call  divisionLoop    ; calling divisionLoop so 3 digits to right of
   decimal point can be calculated
```

```
ret          ; return to main
printResults ENDP

; divisionLoop uses the remainder to calculate 3 numbers to right of
   decimal point. The remainder is multiplied by 10 and then
; divided by numnum. The result is then printed 2 more times.
divisionLoop PROC
xchg  ebx, edx     ; moving remainder to edx, so it can be later be moved
    to eax for div
mov   ecx, 3      ; initializing counter for loop
; division multiplies the remainder by 10 and then divides that by numnum
    to generate the first digit to the right of the decimal point.
; it generates the remaining 2 digits by taking the remainders (from each
    division) and again multiplying them by 10 and, again, dividing
; by numnum. I ultimately realized that I could have just multiplied the
   remainder by 1000 and divided by numnum to get the 3 digits I needed,
; but there were certain situations where that wouldn't work properly (
   specifically if there was no remainder), and I didn't want to code the
    conditionals
; to deal with that after producing this functional code
division:
xchg  eax, edx     ; swapping eax and edx because we're more concerned
   with remainder
xor   edx, edx     ; clearing edx for division
mul   tenten      ; multiplying remainder by 10
div   numnum      ; dividing by original divisor
call  writedec     ; printing digit
loop  division     ; restarting loop (run through loop 3 times in total)
ret          ; return to printResults
divisionLoop ENDP

END main
```